Expression Compatibility Problem

(Extended Version)

Seyed H. HAERI (Hossein)*, Sibylle Schupp

Institute for Software Systems, Hamburg University of Technology (TUHH), Hamburg, Germany

Abstract

We solve the Expression Compatibility Problem (ECP) – a variation of the famous Expression Problem (EP) which, in addition to the classical EP concerns, takes into consideration the replacement, refinement, and borrowing of algebraic datatype (ADT) cases. ECP describes ADT cases as components and promotes ideas from Lightweight Family Polymorphism, Class Sharing, and Expression Families Problem. Our solution is based on a formal model for Component-Based Software Engineering that pertains to the Expression Problem. We provide the syntax, static semantics, and dynamic semantics of our model. We also show that our model can be used to solve the Expression Families Problem as well. Moreover, we show how to embed the model in Scala.

Keywords: Expression Compatibility Problem, Expression Problem, Lightweight Family Polymorphism, Formal Semantics, Component-Based Software Engineering

1. Introduction

Expression Problem (EP) [1, 2, 3] is amongst the most famous problems in the community of Programming Languages (PLs). A wide range of EP solutions have thus far been proposed. Consider [4, 5, 6, 7, 8], to name a few. The essence of EP is the challenge of finding an implementation for an algebraic datatype (ADT) – defined by its cases and the functions on it – that:

- **E1.** is *bidirectionally extensible*, i.e., both new cases and functions can be added.
- **E2.** provides *strong static type safety*, i.e., applying a function f on a statically constructed ADT term t should fail to compile when f does not cover all the cases in t.
- **E3.** upon extension, forces *no manipulation or duplication* to the existing code.
- **E4.** accommodates *separate compilation*, i.e., compiling the extension imposes no requirement for repeating compilation or type checking of existing code. Such static checks should not be deferred to the link or run time either.

*Corresponding author
Email addresses: hossein@tuhh.de (Seyed H. HAERI (Hossein)), schupp@tuhh.de (Sibylle Schupp)

The main reason why EP is peculiarly recurrent in the PLs community is perhaps for it is customary to implement the syntax of a PL as an ADT. Then, the PL semantics as well as the analyses and transformations that pertain to the PL can all comfortably be considered functions on that ADT. As such, EP is prudent about minimising the side effects of **extend**ing an ADT with new cases and functions on it.

On the other hand, family polymorphism [9] and its lightweight variations [10, 11, 12] focus on the software interdependencies in broader terms than PLs and the representing ADTs. They model a group of interdependent software artefacts together as a *family*, with the artefacts themselves being members of the family. The aim here is to keep the interdependencies upon the move to the next software release. Rather than addition of new members, the focus of study in family polymorphism is mostly on the next release **refin**ing (the functionalities of) the existing members to acquire a new family.

It happens that, in addition to being extended by new pieces of syntax and semantics, a PL is also likely to refine some of its constructs over the course of time. It is, of course, of interest in such a situation too for the PL to retain its integrity. Nonetheless, except via nested intersection [13, 14, 15], the connection between EP and family polymorphism has hardly been a source of attraction.

It also turns out that PLs sometimes **replace** older pieces of their syntax and semantics with newer ones. The impacts of such a replacement on the PL soundness constitute a classical source of interest for the PL community [16, §11]. Despite its importance, we are not aware of relevant studies in the field of EP. The following scenario deserves even more attention: PLs are commonly designed by **borrow**ing constructs from a number of other PLs [17]. This is whilst the borrowed constructs are expectably designed only for integration within their original PLs. Institution Theory [18] has historically proved itself as a powerful tool for studying whether the packing of all those constructs together gives rise to a sound combination [19]. Again, the interplay of such a borrowing with EP is, however, an open question.

In this paper, we introduce and solve the *Expression Compatibility Problem* (ECP), which is a variation of EP that also takes the following into consideration: refinement, replacement, and borrowing of PL constructs. Components are a cornerstone to ECP. This is because, in order to ensure soundness properties such as the ones explained above, one needs to enforce constraints at the level of PL constructs. The implication is modelling PL constructs using components in their Component-Based Software Engineering (CBSE) sense.[1] With its above demand for components that are designed for cross-PL reuse, ECP is also a variation of the Expression Families Problem (EFP) [22]. The next paragraph gives an outline of ECP in terms of the EP terminology. For the rest of this paper, we dismiss the PL implementation concerns and employ the EP terminology.

ECP is the challenge of finding an implementation for an ADT that uses components to address EP and:

EC1. is *independently extensible* [23, 5] (a.k.a. *extensionally unifiable* [24]), i.e., it should be possible to compose independently developed extensions.

EC2. is *backward compatible*, i.e., upon extension, the existing code retains its consistency and remains available for use – specially, to the extension[2].

[1] Consult Sommerville [20, §17] and Pressman [21, §10] for comprehensive discussions.

[2] namely, guarantees both *non-destructive* and *object-level* extensibility [13, 8]

EC3. is *scalable* [13], i.e., adding a new case or function takes proportional code.

EC4. ensures *completeness of component composition*, i.e., no valid combination of components should be rejected statically or dynamically.

EC5. *statically* ensures *soundness of component composition*, i.e., provides means to enforce compilation failure for invalid component combinations.

EC6. is *combination sensitive*, i.e., distinguishes between ADT types by their component combinations. In particular, the type identity of a core ADT should be distinguishable from that of its extension.

EC7. *accumulates nominal subtyping*, i.e., recognises the component combination c_1 as a structural subtype of c_2 when every case in c_1 is a nominal subtype of a case in c_2.

EC8. guarantees *syntactic compatibility upon extension*, i.e., statically rejects addition of new cases when the syntactic category of one existing case (or more) is neither retained intact nor refined.

We call the first six the *case concerns* of ECP: those which are concerned with issues about ADT cases. Likewise, we call the first five plus the last two the *function concerns*: those concerned about functions defined on ADTs. That is, the first five are both case concerns and function concerns. One may wonder why EC8 in presence of EC5? Firstly, EC8 is particularly concerned about the sanity of extensions; whereas EC5 is also about core combinations ab initio. Secondly, EC5 is both a case and a function concern whilst EC8 is only a latter.

EC1–EC3 and EC8 underpin new extension concerns w.r.t. EP. Refinement and removal concerns are EC4, EC5, EC6, and EC7. Then, replacement is the focus of EC4, EC5, and EC6.

The list of our major contributions is as follows:

1. We define ECP and present a solution for it in Section 2.
2. In Section 4, we introduce $\gamma\Phi C_0$ as a formal solution to ECP. In summary, $\gamma\Phi C_0$: facilitates minimal shape exposure using its family parameterisation; accommodates component late-binding using a notion component equivalence; and, features static dependency on the 'requires' interface of its components.
3. We show how $\gamma\Phi C_0$ can be used to solve EFP as well. This is done in Section 5 by offering solutions to the two EFP exercises [22].[3]
4. In Section 6, we demonstrate highlights of how to simulate $\gamma\Phi C_0$ in Scala.

Here is how this paper is organised: We begin by Section 2 where the benefits of an ECP solution are highlighted. To put our work in context, we then explore the related work in Section 3. (The reader how is not familiar with that literature and is rather interested in the contents of the current paper might choose to skip over Section 3.) Afterwards, in Section 4, a formal presentation of $\gamma\Phi C_0$ is provided. More particularly, the $\gamma\Phi C_0$ syntax comes in Section 4.1. Selected parts of the $\gamma\Phi C_0$ semantics that are in its front line of solving ECP are discussed in Section 4.2. How solving EFP can be accomplished in $\gamma\Phi C_0$ is, then, shown in Section 5. Next, Section 6 explains how to simulate the $\gamma\Phi C_0$ material of this paper in Scala. Conclusion and future work come

[3] Those exercises are designed for exhibiting the power of Modular Visitor Components (MVCs) for solving EFP.

in Section 7. Finally, Appendix A and Appendix B detail the static and dynamic semantics of $\gamma\Phi C_0$, respectively.

In this paper, we only show how to employ $\gamma\Phi C_0$ for compositional analyses and transformations [25]. For how to employ $\gamma\Phi C_0$ in non-compositional scenarios, see [26, §8]. In addition, we will follow the EP tradition of using basic integer arithmetic as the running example of this paper. To examine the performance of our material under real research languages, the reader is invited to consider [26]. The material that we present in this paper, including both the $\gamma\Phi C_0$ and the Scala code, is available online at: `http://www.sts.tu-harburg.de/people/hossein/ecp.html`.

2. Once upon a time when ECP was already tackled...

This section aims to explore the potential opportunities that a solution to ECP will create for safe reuse. That is accomplished by a rapid presentation of how $\gamma\Phi C_0$ manages to address ECP. On our way, we will introduce the relevant $\gamma\Phi C_0$ features and the techniques needed to get them to serve our purpose.

Components. We begin the show with the introduction of a few ADT cases. $\gamma\Phi C_0$ facilitates that using its special support for *component* definitions:

```
01  component Num<X ◁ Num> {
02    int n;
03    Num(int n) {this.n = n;}
04  }
05  component UNm<X ◁ UNm> {
06    uint n;
07    UNm(uint n) {this.n = n;}
08  }
09  component Add<X ◁ Add> {
10    X left, right;
11    Add(X left, X right) {
12      this.left = left; this.right = right;
13    }
14  }
15  component Sub<X ◁ Num ⊕ Sub> {
16    X left, right;
17    Sub(X left, X right) {
18      this.left = left; this.right = right;
19    }
20  }
```

The code above introduces four components for ADT cases for signed integers, unsigned integers, addition, and subtraction. Those are *Num* (lines 01–04), *UNm* (lines 05–08), *Add* (lines 09–14), and *Sub* (lines 15–20), respectively. The body of a component is like that of a class in Igarashi et al. [27]: Line 02 above states that *Num* has a field n of type int. Line 03 demonstrates *Num*'s constructor. And, although not shown here, $\gamma\Phi C_0$ components can also define (non-constructor) methods.

Unlike an Igarashi et al. class, however, a $\gamma\Phi C_0$ component enjoys *family parameterisation* too. This is for a component to express its 'requires' interface ([20, §17] and [21, §10]). For example, line 15 above states that *Sub* 'requires' for its container ADT to contain *Num* and *Sub*. (In this paper, *Sub* represents signed subtraction, exclusively. Hence, it 'requires' the signed integer case – i.e., *Num* – as well.) Inside *Sub*'s body, the *family parameter* X plays the container role and is to be substituted for a *family*. (See below for more.) Built-in types aside, the only types allowed inside a component body are its family parameter (e.g., lines 10 and 11 above) and the components it 'requires'. (For types of the latter group, see lines 02 and 05 in *EN* and *EU* discussed later on.)

```
01  family  Φ0 = UNm ⊕ Add;          02  family  Φ1 = Num ⊕ Add;
03  family  Φ2 = UNm ⊕ Sub; //Error! Sub requires Num too.
04  family  Φ3 = Num ⊕ Sub;          05  family  Φ4 = Num ⊕ Add ⊕ Sub;
06  family  Φ5 = Sub; //Error! Sub requires Num too.
```

Families, EC4, and EC5. A $\gamma\Phi C_0$ *family* definition can be employed for an ADT introduction. As shown above, a family definition is as simple as listing the components it contains. Families Φ_0, Φ_1, Φ_3, and Φ_4 are examples of $\gamma\Phi C_0$'s support for EC4: The programmer is entitled to mix components in all valid ways. On the contrary, Φ_2 and Φ_5 attempts are statically rejected. Those are examples of $\gamma\Phi C_0$'s support for EC5.

Clients. Functions on ADTs are implemented in $\gamma\Phi C_0$ using its so-called *client* definitions. The two differences between $\gamma\Phi C_0$ components and clients are as follows: Firstly, the former can have fields as well as methods, but, the latter is only a pack of methods. Secondly, the former takes a single family parameter, whilst, the latter can take multiple such parameters. Whilst the first difference is displayed in this section, the second will only be discussed in Section 4. The code below on evaluation of integer arithmetic expressions is an example of how to define functions on ADTs.

```
01  client  EN<X ◁ Num> {//signed integer evaluation
02    int eval(X.Num x, int e(X)){return x.n;}
03  }
04  client  EU<X ◁ UNm> {//unsigned integer evaluation
05    uint eval(X.UNm x, int e(X)){return x.n;}
06  }
```

The most noteworthy point about the two clients above is that they only handle their own part of evaluation. Note that, in order to handle a case, explicit nomination of the case is required. The consequence, which was also stated earlier on, is that: any attempt to handle other cases than the ones nominated in the 'requires' interface will be statically rejected. For example, would it happen that the programmer mistakenly chooses to handle subtraction in *EN*, a compile error will be emitted. The second parameter of *eval* (in lines 02 and 05 above) represents the complete evaluation recipe, the role of which will become more clear below.

```
01  client ENA<X ◁ Num ⊕ Add> {//signed integer and addition evaluation
02    int eval(X x, int e(X)){
03      return x match {
04        case X.Add ⇒ e(x.left) + e(x.right);
05        case X.Num ⇒ EN<X as Num>.eval(x, e);
06      }
07    }
08  }
```

Consider line 04 above, where the joy of compositional evaluation and component-based development are experienced in tandem: Not having the complete recipe, it leaves evaluation of subexpressions to the parameter e. At the same time, only the addition case is handled by that line. The action of line 05 is slightly different, where handling the case (of signed integers) is relayed to the *eval* method of *EN*. More remarkable is the use of family parameter *projection* for the relaying. When not all the components of 'requires'

interface are to be passed in the exact same order, $\gamma\Phi C_0$ demands explicit nomination of the relevant ones. Otherwise, the family parameter alone can be nominated. (See $Eval_1$ and $Eval_4$ below.)

EC7 and EC8. Line 05 above also demonstrates another important property of $\gamma\Phi C_0$, i.e., how it addresses EC7: It is legal for *ENA* to reuse the *EN* methods for its 'requires' interface is a superset of that of *EN*. The dual of that is how the following piece of code fails statically due to $\gamma\Phi C_0$'s implementation of EC8.

```
01  client BadENA<X ◁ Num ⊕ Add> {
02    ... EU<???>.eval(...);//Error! Invalid projection.
03  }
```

The reader is invited to take their time to observe that no valid combination of the components in the 'requires' interface of *BadENA* can be substituted for ??? in line 02.

EC1. Due to its similarity, we drop the definition of $ENS\langle X \lhd Num \oplus Sub\rangle$ to move to the demonstration of how $\gamma\Phi C_0$ addresses EC1 as a function concern. The lines 04 and 05 below show how *ENAS* composes independently developed extensions of *EN*, i.e., *ENA* and *ENS*. (It is trivial to observe that $\gamma\Phi C_0$'s family definition facility addresses EC1 as a case concern.)

```
01  client  ENAS<X ◁ Num ⊕ Add ⊕ Sub> {
02    int eval(X x, int e(X)){
03      return x match {
04        case X.Add ⇒ ENA<X as Num ⊕ Add>.eval(x, e);
05        case X.Sub ⇒ ENS<X as Num ⊕ Sub>.eval(x, e);
06        case X.Num ⇒ EN<X as Num>.eval(x, e);
07      }
08    }
09  }
```

Tests. In order to bring the above clients to action for the defined families, one first ties the recursive knot. $Eval_1$ and $Eval_4$ below demonstrate that for Φ_1 and Φ_4, respectively:

```
01  client  Eval1<X ◁ Num ⊕ Add>{
02    int do_it(X x){
03      return ENA<X>.eval(x, do_it);
04    }
05  }
```

```
06  client  Eval4<X ◁ Num ⊕ Add ⊕ Sub>{
07    int do_it(X x){
08      return ENAS<X>.eval(x, do_it);
09    }
10  }
```

Then, one instantiates the clients using the families defined earlier. Here are some tests:

```
01  val tpf_1 = new Add<Φ1>(new Num<Φ1>(3), new Num<Φ1>(5));
02  val tpf_3 = new Add<Φ3>(new Num<Φ3>(3), new Num<Φ3>(5));
↪     //Error! Φ3 doesn't provide Add.
03  Eval_1<Φ1>.do_it(tpf_1); //Returns 8.
04  val tpf_4 = ... //3 + 5 for Φ4
05  val tpfmo_4 = new Sub<Φ4>(tpf_4, new Num<Φ4>(1));
06  Eval_4<Φ4>.do_it(tpf_4); //Returns 8.
07  Eval_1<Φ4>.do_it(tpf_4); //Error! Sub case of Φ4 unmatched by Eval_1.
08  Eval_4<Φ4>.do_it(tpfmo_4); //Returns 7.
09  Eval_1<Φ4>.do_it(tpfmo_4); //Error! Sub case of Φ4 unmatched by Eval_1.
10  Eval_1<Φ4 as Add ⊕ Num>.do_it(tpf_4); //Returns 8.
11  Eval_1<Φ4 as Add ⊕ Num>.do_it(tpfmo_4);
↪     //Error! tpfmo_4 contains other Φ4 cases than Num and Add.
```

Other ECP Concerns. We end by a short explanation on how $\gamma\Phi C_0$ addresses the remaining ECP concerns. The support for EC2 is obvious for addition of new components and clients has no effect on existing $\gamma\Phi C_0$ programs. $\gamma\Phi C_0$ does also clearly support EC3: Addition of a new case amounts to defining the respective single component; addition of a new function on n cases (say for pretty-printing) amounts to defining (at most) n new clients for the corresponding cases and a single client to tie the recursive knot. Understanding that $\gamma\Phi C_0$ addresses EC6 is also easy: Defining a new ADT (say as an extension to an existing one) is a matter of defining a new $\gamma\Phi C_0$ family. Such an addition influences no existing family (especially that of the core ADT, if any). A family is distinguishable from other families (including its core, if any) by its name. On the contrary, clients are equally (un-)available to families with identical component combinations.

Remark 1. Note that, although after tying the knot for $Eval_1$ or $Eval_4$, they are no longer extensible, that is not a failure in addressing EP. The reason is that the evaluation service that was available to Φ_1 and Φ_4 essentially come from *ENA* and *ENAS*, which remain extensible upon tying the knot.

Remark 2. The clients presented in this section perform the evaluation task in a component-based fashion. The technique we used to that end is the additive [20, §17.3] flavour of what we call *integration of a decentralised pattern matching* [26, §9]: Instead of centralising the pattern matching in a single place, we distribute it amongst clients that correspond to the cases of the task. This section, nonetheless, was rather focused on the ECP concerns. We postpone dedicated explanation until Section 5 where a clearer account of the technique is developed.

3. Related Work

This section classifies the relevant literature according to their proximity to this paper. Each classification ends in a summary of how they are similar to or different from our work. Inside a classification, an outline of each work is accompanied by a short description of which ECP concerns they manage or fail to score.

Family Polymorphism

Family Polymorphism [9] aims to ensure that certain interrelationships between the constituents of a system – referred altogether as a *family* – are all extension-invariant. Family Polymorphism is a meritorious tool when families with identically-named members that are of identical relative types are to be distinguished from one another. But, otherwise, sharing classes amongst distinct families (with even identically-named members of identical relative types) is notoriously challenging [28]. EC1 and EC2 have clear support in Family Polymorphism. Given its difficulties in cross-family class sharing, whether Family Polymorphism can also support EC3 is hard to judge. In order to examine whether the other ECP concerns can also be addressed, one needs to first decide on the counterpart Family Polymorphism offers for components. The immediate candidate is the *pattern* concept of gbeta [29] that generalises the familiar OOP class concept. One might manage to employ *virtual constraints* [30] to get the pattern concept to simulate a component. (The simulation will perhaps be accomplished like how Scala traits and type constraints are employed for the same purpose. See Section 6.) Given the acquired taste of programming in gbeta, however, we did not try that simulation. We, hence, cannot assess how addressable the remaining ECP concerns are by Family Polymorphism.

Class Sharing

Jx [31] starts a series of works on sharing nested classes amongst families. Jx itself only allows that amongst hierarchies of families. J& [13] generalises that to intersection types [32] to cater extension composition (cf. EC1). Of the ECP concerns, it also addresses EC3 (using late-binding) and EC2. In $J\&_s$ [14], sharing of classes amongst families is nomination-based but not restricted to those which are hierarchically related. $J\&_h$ [15] moves to **homogeneous** class sharing, where a core family and all its extensions are *equivalent*. To that end, when a new family extends a core family, $J\&_h$ automatically updates all the equivalent families (including the core itself) with the extensions provided by the new family. Such an update is likely to risk EC2.

The similarity between nested family classes and our components makes $J\&_s$ particularly close to our work. Yet, there are at least three significant differences between this entire thread of research and that of ours: First, the sharing of components between a base family and its extensions is not limited in our approach to identical components; substitution for equivalent ones is equally acceptable (cf. Remark 10). Second, in their research thread, the shared classes have tight dependencies to their enclosing families (hierarchically, heterogeneously, or homogeneously). That is, addition of a new class (for sharing) amounts to addition of a new family or updating an existing one. In either case, (re)compilation of an enclosing family is inevitable. Our components are, however, not dependent on any family. (In fact, they are even not enclosed by a family.) Addition of a new component implies compilation of nothing but the component itself. Third, with their independent nature, components can enforce constraints on the families they are chosen for inclusion in. Shared classes, on the other hand, have no control over that to the degree that they can be shared via arbitrary enclosing families (subject to certain workflow disciplines that are out of scope here).

Lightweight Family Polymorphism

Having observed the heavy workload of family polymorphism in certain circumstances, Saito, Igarashi, and Viroli [10] craft lightweight family polymorphism. Similar to family

polymorphism, they target extensibility of mutually recursive classes. Yet, inspired by Jolly et al. [33], they adopt a "classes-as-families" principle. Saito, Igarashi, and Viroli develop a formal model .FJ for lightweight family polymorphism, for the type system of which they also prove an important correctness theorem. In .FJ, families and their members are represented using top-level and nested classes, respectively. In $\gamma\Phi C_0$, however, both families and components are top-level. An "inheritance-without-subtyping" approach is taken in .FJ for family members. That is, whilst a same-named nested class of an extending family does (automatically) inherit from that of the base family, there is no subtyping relationship between the two nested member classes. We take a different approach: $\gamma\Phi C_0$ components that are common between two families are indeed the same (and, hence, in a subtyping relationship). Finally, .FJ addresses EC2 and EC3. It fails to address EC1 because it only supports extension using single (nominal) inheritance of top-level families. For its lack of counterpart for components, the rest of ECP concerns are irrelevant to .FJ.

Kamina and Tamai [11] notice the monolithic nature of (lightweight) family polymorphism in that family members are inseparable from their enclosing families. They craft FGJ# to get what they call *type parameter members*. Using this addition, they can take interfaces of family members out of family bodies so that members are no longer nested classes of families. Consequently, families and their members gain independence to the level of separate compilation from each other. However, addition of a new member is still not possible without first providing a new family interface that adds a new type parameter to correspond to the new member. It is only thereafter that one is authorised to refer to the new member using their type parameter member notation. Kamina and Tamai do not discuss whether reuse of the FGJ# code that works for a base family is valid for an extension that adds new members. Hence, whether FGJ# supports EC4, EC7, and EC8 is not known. On the other hand, for similar reasons to .FJ, FGJ# addresses EC2 and EC3 but not EC1. Its move towards components shapes an interesting behaviour w.r.t. the rest of ECP concerns. It can simulate them all so long as an ADT interface hard-wires the relevant requirements.

Lightweight dependent classes [12] is the next work of Kamina and Tamai, with remarkable proximity to $\gamma\Phi C_0$. Aiming at reducing the boilerplate code of the above two works of the section, they take one further step toward getting component-based. They provide X.FGJ consisting of member interfaces and families. In X.FGJ, a family *materialise*s its members by providing nested classes with empty bodies that derive from their respective member interfaces. Besides, by abstracting members to the level of member interfaces, defining extensions which refine existing ADT cases is rather straightforward. Despite all its improvements, in X.FGJ too, defining new members is only possible in tandem with defining new families which materialise the respective new member interfaces. Interestingly enough, this is due to the fact that in X.FGJ, members are **nested** classes of their enclosing families. On the contrary, $\gamma\Phi C_0$ components are completely detached from the families they are to be members of. Our final remark about X.FGJ is that it scores similar to FGJ# regarding the ECP concerns.

Whilst the works in this group are particularly sophisticated on case refinement, they are almost silent about addition of new cases. ECP, however, is concerned about both sorts of extension. That is also why $\gamma\Phi C_0$ provides direct means to address both.

Expression Families Problem

Oliveira was the first to frame EFP. He also offered MVCs [22] as an EFP solution. Oliveira and Cook [6] back this work up by the powerful and simple concept of *object algebras* [34]. Later, Oliveira et al. [35] try to address awkwardness issues faced in their former paper upon composition of object algebras. As detailed by Haeri [26], which element in the latter two works to compare against $\gamma\Phi C_0$ components (and families) is not straightforward to us. With that difficulty in assessment, we only concentrate on Oliveira's seminal work in what follows.

MVCs address all EC concerns except EC3, EC5, and EC8. Their support for EC7 is worth a special attention: In his machinery, Oliveira chooses an extension to be a *super*type of a core. Whilst the reason for that choice is not clear to us, we wonder whether the consequence is that they support EC7 in the same direction of ours (c_1 subtype of c_2) or the opposite one. Like Zenger and Odersky [36] – and in line with the entire OOP literature – we choose an extension to be a *sub*type of a core ADT. This is a special case of our EC7.

Expression Problem

Of the rich literature on EP we only consider a couple that we deem close enough to this paper. Garrigue [37] solves EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. Per se, a function that pattern matches on a group of these global cases can serve any ADT containing the selected group. Garrigue's work is based on OCaml's built-in support for Polymorphic Variants [38] and addresses all the ECP concerns except EC5 and EC6. Yet, from ECP's standpoint, the most important missing factor is a notion of components in this work. Rompf and Odersky [39] employ a fruitful combination of the Scala features to present a very simple yet effective solution to EP using LMS. The support of LMS for E2 can be easily broken using an incomplete pattern matching. Yet, given that pattern matching is dynamic, whether LMS really relaxes E2 is debatable. Although LMS is not based on components, it scores all the ECs except EC5, EC6, and EC8.

4. $\gamma\Phi C_0$: A Formal Solution to ECP

As a PL, $\gamma\Phi C_0$ is highly inspired by the popular informal models of CBSE. Most remarkably, that includes the works of Pressman [21, §10] and Sommerville [20, §17]. $\gamma\Phi C_0$ is, however, especially tuned for ECP. The opening discussion of this section will delve into the design choices made accordingly. From a PL designer's perspective, on the other hand, one would be interested in finding out about the axes along which to tweak the PL for reuse. In the favour of that interest, over the opening discussion, we further elaborate on the two sources of polymorphism in $\gamma\Phi C_0$. After the opening discussion, Section 4.1 presents the $\gamma\Phi C_0$ syntax. Next, in Section 4.2, we will focus on the most immediately relevant parts of the $\gamma\Phi C_0$ semantics to the ECP concerns. For the complete $\gamma\Phi C_0$ semantics, see Appendix A and Appendix B. We now move to the opening discussion itself.

The $\gamma\Phi C_0$ world has three major role-players: components, families, and clients. A $\gamma\Phi C_0$ component takes after its well-known CBSE identity by specifying its 'requires' and 'provides' interfaces. It specifies its 'requires' interface using family parameterisation.

The 'provides' interface of a $\gamma\Phi C_0$ component is specified just like a familiar OOP class. $\gamma\Phi C_0$ families are simply defined as a collection of their respective components. Clients in $\gamma\Phi C_0$ are pieces of code that are applicable to any family, provided that the family contains the specified minimal component combination.[4] (See Figure A.11.) This is the first source of polymorphism in $\gamma\Phi C_0$: A client (or component) code can be reused amongst all such families (but, not other ones). We refer to this property as the *minimal shape exposure*. Again, $\gamma\Phi C_0$ uses family parameterisation to that end. Minimal shape exposure is particularly geared towards EC4, EC5, and EC6.

Polymorphism in $\gamma\Phi C_0$ comes from another source as well: *component late-binding*. The family parameterisation used for a $\gamma\Phi C_0$ component or client enforces the availability of certain components. Instead of providing the exact requested components, however, the family is free to mix an *equivalent* component in. (See (S-WFP) and (S-PFP) in Figure A.12.) As such, the exact behaviour of a family-parameterised code is not known until the actual family used for instantiation is provided. This is our notion of component late-binding. This source of polymorphism serves EC7 and EC8 most specifically.

In the presence of the above two sorts of polymorphism, $\gamma\Phi C_0$ components, clients, and their methods are all chosen to be *type*-monomorphic. Type polymorphism is already well researched, and, we see little extra service that type-polymorphic elements can bring to ECP, if any. For similar reasons, $\gamma\Phi C_0$ is designed with no inheritance between components or clients. Furthermore, with our lack of applications where a component ought to take multiple family parameters, $\gamma\Phi C_0$ chooses components to take a single family parameter. $\gamma\Phi C_0$ clients, on the other hand, are eligible to take more than one family parameter.[5]

By specifying its 'requires' interface using family parameterisation, a $\gamma\Phi C_0$ client (or component) determines the component combination it expects from the family that is going to use it. The client (or component) code, then, will be statically bound to the requested combination. (See (WF-VCase) in Figure A.10 and (T-Test) in Figure A.14.) Inside the body of a client (or component), any attempt to access unrequested components is outlawed statically. We refer to this feature as static dependency on the 'requires' interface.

For the rest of this paper, we maintain the following conventions: Unless otherwise stated, when we refer to components, families, and clients, we mean the $\gamma\Phi C_0$ ones. Moreover,"_" is our wildcard; it shows our lack of interest in the exact piece of information that "_" is used in place of.

Remark 3. Unlike the code snippets of Sections 2 and 5, the formal model that is presented in this section for $\gamma\Phi C_0$ does not include the following features: component-based pattern matching (e.g., lines 03–07 of *ENAS*), built-in types (e.g., line 02 of *Num*), functions-as-arguments (e.g., line 02 of *EN*), and top-level variables (e.g., tpf_1 in Section 2.). Our impression is that, although programmatically valuable, the lack of those features is no loss for the formal model as a core calculus.

[4] As such, a client is like a set of family-polymorphic methods in its .FJ [10] sense. The main difference is that .FJ enforces inheritance from a top-level family, whilst $\gamma\Phi C_0$ enforces the availability of certain components (and absence of the others).

[5] All that simplification together might of course become deceptive about $\gamma\Phi C_0$. We would, therefore, like to invite the reader to investigate the power of $\gamma\Phi C_0$ in Sections 5 where long-standing research challenges are easily taken up.

Remark 4 (CBMCalc Comparison). $\gamma\Phi C_0$ is, in fact, a successor of CBMCalc [26]. The main differences between the two are that $\gamma\Phi C_0$

1. gives EC1 a first-class support using its simple mechanism for cross-client method calls. CBMCalc can only simulate that using linearisation of multiple inheritance.
2. ships with no inheritance for components or clients. CBMCalc ships with inheritance for both.
3. is designed not to include super calls. This is a consequence of observing the absence of a mixin flavour in CBMCalc's use of inheritance.
4. allows clients to take more than one family parameter.
5. also models Common Reuse Principle [40] for components. (cf. Remark 5.)

4.1. The $\gamma\Phi C_0$ Syntax

Figure 1 shows the syntax of $\gamma\Phi C_0$. Here, γ ranges over components, C over clients, K over constructors, m over methods, x over method parameters, e over expressions, f over fields, Φ over families, and X over family parameters. We take this to be a reserved variable name. Priming a syntactic metavariable or subscripting it with numbers does not change its syntactic category. When distinction between a metavariable of a component and that of a client is required, we use subscripts γ and C. When referring to both categories collectively, we drop the subscript. For example, m_γ and m_C denote component methods and client methods, respectively, but m can be either an m_γ or an m_C. Note that neither subscript is related to a particular component γ or client C.

The syntax in Figure 1 is divided into two parts: relative and absolute. The former syntax is all relative to the family parameters and is about before substitution of (real) families for the family parameters. The latter, on the other hand, is regarding after family introduction, wiz., when some defined families get employed for substitution for the family parameters of the relative syntax.

Following the tradition of lightweight family polymorphism (Section 3), the overline notation is used for a list of entities and $\#(.)$ for the length of a list. For example, for some known n: $\overline{\gamma}$ denotes $\gamma_1\gamma_2\ldots\gamma_n$, and, $\mathsf{this}.\overline{f} = \overline{f}$ denotes $\mathsf{this}.f_1 = f_1, \ldots, \mathsf{this}.f_n = f_n$. The notation $\overline{\gamma}$ is also overloaded to mean the list $\gamma_1, \gamma_2, \ldots, \gamma_n$, when appropriate. We extend that notation for $\oplus\overline{\gamma}$ to mean $\gamma_1 \oplus \gamma_2 \oplus \ldots \gamma_n$. We overload the notation one further step to mean $X_1 \lhd \oplus\overline{\gamma}_1, X_2 \lhd \oplus\overline{\gamma}_2, \ldots, X_n \lhd \oplus\overline{\gamma}_n$ by $\overline{X} \lhd \overline{\oplus\overline{\gamma}}$. We use ϵ for empty lists.

Using the relative syntax, one can define components (γ) and clients (C). Both clients and components take family parameters. The notation $X \lhd \oplus\overline{\gamma}$ stipulates that the family to be substituted for X has to include components (that are equivalent to) $\gamma_1, \gamma_2, \ldots, \gamma_n$. In such a case, we say that $\oplus\overline{\gamma}$ is the *upper bound* of X. Similarly, we say that γ_i is *an* upper bound of X, for $i = 1, 2, \ldots, n$. We call n the upper bound size of X – written as $\#X = n$ – when $X \lhd \oplus\overline{\gamma}$ and $\#\oplus\overline{\gamma} = n$. When γ is an upper bound of X, the relative type $X.\gamma$ – read the component γ of X – is the component substituted for γ when substituting a family for X.

When a method of a client G calls a method of another client G', the caller needs to state the family parameter(s) of G to be used for substitution for the family parameter(s) of G'. In such a call, when the family parameter X of G is to be used for X' of G', two possibilities are available: $G'\!<\cdots, X, \cdots\!>$ and $G'\!<\cdots, X \ \mathsf{as} \ \oplus\overline{\gamma}, \cdots\!>$. In the first case, the same family eventually substituted for X in G will be substituted intact for X' in

Relatives

D_γ	::= component γ<$X \lhd \oplus\overline{\gamma} \not\lhd \overline{\gamma}$> $\lhd \{\overline{R}\,\overline{f}\,;\ K\ \overline{M_\gamma}\}$	component definition
D_C	::= client C<$\overline{X}\ \lhd \overline{\oplus\overline{\gamma}}$>$\{\overline{M_C}\}$	client definition
G	::= $\gamma \mid C$	relative definition name
R	::= $X \mid X.\gamma$	relative type
S	::= $X \mid X$ as $\oplus\overline{\gamma}$	family parameter selection
K	::= $\gamma(\overline{R f})\{$this.$\overline{f} = \overline{f}\,;\}$	constructor
M_γ	::= $R\ m_\gamma(\overline{R}\ \overline{x})\{$return $e_r\,;\}$	component method
M_C	::= $R\ m_C(\overline{R}\ \overline{x})\{$return $e_C\,;\}$	client method
e_r	::= $x \mid e_r.f \mid e_r.m_\gamma(\overline{e_r}) \mid$ new $X.\gamma(\overline{e_r})$	relative expression
e_C	::= $e_r \mid m_C(\overline{e_C}) \mid C$<$\overline{S}$>$.m_C(\overline{e_C})$	client expression

Absolutes

D_Φ	::=	family $\Phi = \oplus\overline{\gamma}$	family definition
Z	::=	$\Phi \mid \Phi$ as $\oplus\overline{\gamma}$	family selection
A	::=	$Z \mid \gamma$<Z>	absolute type
I	::=	C<$\overline{Z}$>	client instance
τ	::=	$\overline{D_\Phi}\,; I.m_C(\overline{v})$	test
e_a	::=	$e_a.f \mid e_a.m_\gamma(\overline{e_a}) \mid$ new γ<Z>$(\overline{e_a}) \mid I.m_C(\overline{e_a})$	absolute expression
v	::=	new γ<Z>$(\overline{v})$	value

Figure 1: The $\gamma\Phi C_0$ Syntax

G'. The second case signifies the situation when only the selected components $\overline{\gamma}$ of the family substituted for X in G will substituted for X' in G'. We say, in such a case, that X is *projected* to $\oplus\overline{\gamma}$. Additionally, we call X as $\oplus\overline{\gamma}$ a *family parameter projection*. The projection of a family parameter is a must when $\#X' \neq \#X$.[6] That manifests the role of S in Figure 1. Z plays a similar role for the absolute syntax.

The syntax for introduction of a family is as simple as enumerating the components it combines, interleaved by "$\oplus$". To test a client on a family, one calls a method of a client by passing appropriate arguments to the method. To that end, either the whole family is used for client instantiation (C<$\cdots, \Phi, \cdots$>) or a *projection* of it (C<$\cdots, \Phi$ as $\oplus\overline{\gamma}, \cdots$>). Note that, unlike components, we choose clients to store no data and simply act as a collection of methods. As such, clients need no constructors.

A test τ contains a sequence of family introductions and a single call to a method of a client instantiated by the introduced families (or their projections). A $\gamma\Phi C_0$ program is a test along with the components and clients that it uses. For a $\gamma\Phi C_0$ program to compile and run, we assume the availability of two more ingredients: First, a customary *class table* CT, which is a simple mapping from the names of the program components, clients, and families to their bodies. We assume similar sanity conditions for CT to

[6] This syntax is also meaningful when the order of components to be passed to G' is not exactly the same as their order of appearance in G's definition. Yet, the discussion in this paragraph is not particularly concerned about the order in which components are enumerated. The order becomes significant in the subsequent sections when **same-indexed**ness is assumed. See Figures A.5, A.9, and A.11 as well.

those assumed by Saito et al. [10]. Second, an equivalence relation $\overset{s}{\equiv}$ as a repository for the components to be considered equivalent. (More on the $\overset{s}{\equiv}$ relation of $\gamma\Phi C_0$ in Remark 10.)

When the premises of a $\gamma\Phi C_0$ rule contain a statement of the form $\gamma\{\cdots\}$, we are abbreviating $CT(\gamma) = \mathsf{component}\ \gamma\{\cdots\}$. The situation is similar for clients. Likewise, $\Phi = \oplus\overline{\gamma}$ abbreviates $CT(\Phi) = \mathsf{family}\ \Phi = \oplus\overline{\gamma}$. With such abbreviations, we drop the explicit mention of CT over $\gamma\Phi C_0$ rules.

Remark 5. Note that, for a definition $\mathsf{component}\ \gamma\texttt{<}X \lhd \oplus\overline{\gamma} \not\lhd \cdots\texttt{>}\{\cdots\}$, it is valid that $\gamma' \in \overline{\gamma}$ whilst $\gamma \neq \gamma'$. That is, a $\gamma\Phi C_0$ component can enforce the availability of other components than itself. This is particularly useful for implementing The Common Closure Principle (CCP) of Martin [40]: "Classes that change together belong together." Similarly, the "$\not\lhd\ \overline{\gamma}$" part of the D_γ notation is designed to implement The Common Reuse Principle (CRP) of Martin [40]: "Classes that aren't reused together should not be grouped together." The conflict between CCP and CRP is, of course, semantically wrong in that classes cannot both change together and not be reused together. That is taken care of by (NC-Comp) in our semantics by refusing the presence of a component in both the "$\lhd \oplus\overline{\gamma}$" part and the "$\not\lhd\ \overline{\gamma}$" part of a single D_Φ (cf. Figure A.9). More general family parameterisation that can describe arbitrary relationships between components can soon slip into computability difficulties that we would like to avoid.

4.2. How does $\gamma\Phi C_0$ address ECP?

This section gives an overview of the static semantics parts of $\gamma\Phi C_0$ that help it solve ECP. Given that EC3 is only quantitative, this section will not take that concern into account. On its way, this section first explains the three most directly related rules of the $\gamma\Phi C_0$ semantics. Then, it proceeds by discussing how each ECP concern is addressed by which of the three rules.

$$\frac{\begin{array}{c}\Gamma; C \vdash \overline{e} : \overline{R_e} \qquad \mathit{fps}(C') = \overline{X'} \qquad \#\overline{X'} = \#\overline{S} \\ C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')\ \mathsf{ok} \qquad \mathit{mtype}(m_C, C') = \overline{R'} \to R' \\ \overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C'] \qquad C \vdash \overline{R_e}\texttt{<:}\overline{R} \qquad R = R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']\end{array}}{\Gamma; C \vdash C'\texttt{<}\overline{S}\texttt{>}.m_C(\overline{e}) : R}\ (\text{T-Invk}_3)$$

$$\frac{\overline{\gamma} = \mathit{fpub}^*(\Phi) \qquad \mathit{sat\text{-}by}(\overline{\gamma}, \Phi) \qquad \mathit{non\text{-}conf}(\overline{\gamma})}{\mathsf{family}\ \Phi = \cdots\ \mathsf{ok}}\ (\text{WF-Family})$$

$$\frac{\begin{array}{c}\overline{D_\Phi}\ \mathsf{ok} \qquad I\ \mathsf{ok} \qquad \mathit{mtype}(m_C, I) = \overline{A} \to A \\ \overline{Z} = \mathit{fsel}(\overline{v}) \qquad I = C\texttt{<}\overline{Z'}\texttt{>} \qquad \overline{Z} = \overline{Z'} \\ \mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{v} : \overline{A'} \qquad \mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{A'}\texttt{<:}\overline{A}\end{array}}{\overline{D_\Phi}; I.m_C(\overline{v}) : A}\ (\text{T-Test})$$

Figure 2: The Rules of $\gamma\Phi C_0$ that Most-Directly Address ECP

Figure 2 illustrates the three rules mentioned above. The clauses used in those rules are likely to look extraordinarily compact to the untrained eye. Before we get into the

intuition behind each rule, we would like to invite the reader to check Figure 3 for the abbreviations used in Figure 2. Likewise, Figure 4 expands on the names of the helper functions used in Figure 2. The exact definitions of the helper functions can be found in Appendix A. We now continue to the rules.

With its number of premises the rule (T-INVK$_3$) might look daunting at the first sight. Below, we explain it informally in a top-down and left-to-right fashion. The rule states that, within the body of a client C, in order for a call to a method m_C of C' to return a value of type R: The argument types $\overline{R_e}$ need to be determined ($\Gamma; C \vdash \overline{e} : \overline{R_e}$); the passed selections $\overline{S}$ to C' need to have the right number ($\mathit{fps}(C') = \overline{X'}$ and $\#\overline{X'} = \#\overline{S}$); the pass of all selections $\overline{S}$ from C to their same-indexed family parameter amongst $\overline{X'}$ of C' needs to be valid ($C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok); the parameter types $\overline{R'}$ of m_C in C' need to be adapted to C for $\overline{R}$ to be obtained ($\overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$); the argument types $\overline{R_e}$ need to all be subtypes of (or equal to) their same-indexed $\overline{R}$ type ($C \vdash \overline{R_e} <: \overline{R}$); and, the return type R' of m_C in C' needs to be adapted for R to be obtained ($R = R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$).

Abbreviation	Stands For
$\Gamma; C \vdash \overline{e} : \overline{R_e}$	$\Gamma; C \vdash e_1 : R_{e_1}, \ldots, \Gamma; C \vdash e_n : R_{e_n}$
$C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok	$C \vdash \mathit{ssfp}(S_1, X'_1, C')$ ok, $\ldots, C \vdash \mathit{ssfp}(S_n, X'_n, C')$ ok
$C \vdash \overline{R_e} <: \overline{R}$	$C \vdash R_{e_1} <: R_1, \ldots, C \vdash R_{e_n} <: R_n$
$R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$	$((R'[C \xrightarrow{[S_1/X'_1]} C']) \cdots [C \xrightarrow{[S_n/X'_n]} C'])$
$\overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$	$R_1 = R'_1[C \xrightarrow{[\overline{S}/\overline{X'}]} C'], \ldots, R_n = R'_n[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$
$\overline{Z} = \mathit{fsel}(\overline{v})$	$Z_1 = \mathit{fsel}(v_1), \ldots, Z_n = \mathit{fsel}(v_n)$
$\mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{v} : \overline{A'}$	$\mathit{as\text{-}fam}(Z_1) \vdash v_1 : A'_1, \ldots, \mathit{as\text{-}fam}(Z_n) \vdash v_n : A'_n$
$\mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{A'} <: \overline{A}$	$\mathit{as\text{-}fam}(Z_1) \vdash A'_1 <: A_1, \ldots, \mathit{as\text{-}fam}(Z_n) \vdash A'_n <: A_n$

Figure 3: Abbreviations Used in Figure 2

Name	Stands For
fps	*family parameters*
ssfp	*substitution* of family parameter *selection* for *family parameter*
mtype	*method type*
fpub	*family parameter upper bound*
sat-by	*satisfied by*
non-conf	*non-conflicting*
fsel	*family selection*
as-fam	*as* a *family*

Figure 4: Helper Functions in Figure 2

The rule (WF-FAMILY) states that for a family Φ to be well-formed: The components $\overline{\gamma}$ that are directly or indirectly requested by the components combined to obtain Φ – i.e., $\mathit{fpub}^*(\Phi)$ – must all (either themselves or an equivalent of theirs) exist in Φ ($\mathit{sat\text{-}by}(\overline{\gamma}, \Phi)$); and, that there must be no conflict between the $\overline{\gamma}$ (namely, $\mathit{non\text{-}conf}(\overline{\gamma})$).

Finally, the rule (T-Test) reads as follows: For a test to take the type A, the families defined in it, as well as its client instance I should be well-defined; the values $\overline{v}$ passed to the client method need to be of same family selections as those used for the client instantiation ($\overline{Z} = \overline{Z'}$); and, the types $\overline{A'}$ of the values need to be subtypes of (or of the same type as) the types $\overline{A}$ that the method called on I accepts ($\mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{A'} <: \overline{A}$).

Armed with the above three rules, below we sketch how $\gamma\Phi C_0$ addresses the promised ECP concerns. For more details, see Appendix A for the precise definitions of the clauses used in the premises of Figure 2.

EC1. As exemplified by *ENAS*, this concern is addressed using calls to methods of other clients. The rule for such a call is (T-Invk$_3$) in Figure 2. In that rule, the only clients involved are the caller client and the callee one. No assumption is made about any (dependent development of any) other client.

EC2. As explained in Section 4.1, a $\gamma\Phi C_0$ program is a test along with the components and clients used. The typing rule of a test is (T-Test) in Figure 2. In that rule, the only components and clients that are involved in the decision making are the ones already used in I. That is C itself and the components used in $\overline{Z}$. In other words, addition of new components or clients is of no influence on the rule. Likewise is the addition of new (well-formed) families. Note that, in the typing and subtyping judgements of (T-Test) – i.e., clauses in the bottom line of its premises – each judgement takes place inside its own family, exclusively.

EC4. The respective rule of this concern is (WF-Family), with the key role player being $\mathit{sat\text{-}by}(\overline{\gamma}, \Phi)$. This clause pronounces Φ sound so long as it does provide all the requested components (or equivalents of theirs).

EC5. Again, the respective rule here is (WF-Family). Albeit, the role of $\mathit{non\text{-}conf}(\overline{\gamma})$ is more important for this concern. This clause makes sure that there is no conflict between the components combined to produce a family.

EC6. The fact that a family is distinguishable from other families is clear. After all, a family is distinguished by its name. Moreover, the same clients are equally available to families with the same component combinations. That can be realised by observing it that every clause in (T-Test) – except $\overline{D_\Phi}$ ok that is irrelevant here – works equivalently for differently named families with the same combination.

EC7. The respective rule of this concern is (T-Invk$_3$). It is the $C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok clause in the premises of that rule that is the most important here. The explicit correspondence between $\overline{S}$ and $\overline{X'}$ is the evidence submitted (by the programmer) for C to be a structural subtype of C'. As such, the reuse of the method m_C of C' by C is authorised by that clause. For every family parameter X'_i of C', the above clause checks whether the substitution of the selection S_i of C is valid for X'_i of C'.

EC8. The respective rule and key clause of this concern are the same as those of the previous concern. In order to explain how this concern is addressed, however, we need to zoom into $C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok. Generally, a clause $C \vdash \mathit{ssfp}(X \ \mathsf{as} \ \oplus\overline{\gamma}, X, C')$ ok uses a premise $\mathit{valid\text{-}as}(C, X, \oplus\overline{\gamma})$ that ensures the following: Projection of a family parameter X of C to $\oplus\overline{\gamma}$ is valid. As such, it will reject the reuse of the method m_C of C' otherwise.

5. Expression Families Problem

In his seminal work on EFP [22], Oliveira provides two self-contained examples that obviate the necessity of solving EFP and the strength of MVCs: Equality Tests and Narrowing. We claimed in the Introduction that ECP is a variation of EFP. (See Section 3 for the comparison on dealing with aggregate subtyping.) The first goal of this section is to substantiate that claim by offering $\gamma\Phi C_0$ solutions to Equality Tests and Narrowing in Sections 5.1 and 5.2, respectively. The second goal of this section is to get the reader to observe how straightforward of a solution to EFP a solution to ECP is. (See Remark 8 for more.) In addition, as stated in Remark 2, the solutions in this section are based on integration of a decentralised pattern matching. Our third goal in this section is to offer a devoted presentation of that technique.

5.1. Equality

The purpose of the Equality Test exercise is to provide a statically safe solution for *multiple dispatching* that is also extensible. Like that of Oliveira, we offer a solution that simulates multi-methods [41, 42]. The driving example is structural equality between expressions. The test on *Num*, *Add*, and *Sub* is performed as follows:

$$\begin{aligned}
equal(Num(n), Num(n')) &= (n == n') && (1)\\
equal(Add(e_1, e_2), Add(e_1', e_2')) &= equal(e_1, e_1') \wedge equal(e_2, e_2') && (2)\\
equal(Sub(e_1, e_2), Sub(e_1', e_2')) &= equal(e_1, e_1') \wedge equal(e_2, e_2') && (3)\\
equal(_, _) &= \bot. && (4)
\end{aligned}$$

Such a formula is typically implemented as a pattern matching on the two expressions together. Using a familiar pattern matching, the implementation will, however, lose extensibility. To prevent that loss, our solution is first to **decentralise** the pattern matching by implementing each case using a single client. Then, one **integrate**s as many of such clients as appropriate in another client, which also ties the recursive knot. *EqNum* and *EqAdd* below are the equality clients that correspond to Equations (1) and (2), respectively:

```
01  client EqNum<X ◁ Num> {
02    bool eq(X.Num x1, X.Num x2, bool e(X, X)){return x1.n == x2.n;}
03  }
04  client EqAdd<X ◁ Add> {
05    bool eq(X.Add x1, X.Add x2, bool e(X, X)) {
06      return e(x1.left, x2.left) && e(x1.right, x2.right);
07    }
08  }
```

Note that both *EqNum* and *EqAdd* only take care of their own part of decentralisation. In a similar manner, one implements *EqSub*<$X \lhd Sub \oplus Num$> and *EqDef*<$X \lhd \epsilon$> for Equations (3) and (4). Due to minimal shape exposure of X, none of the above four equality clients sees more than its pertinent part of the recipe. Nevertheless, for usage at the appropriate point (e.g., line 6 above), provisional access to the complete recipe is granted to them via their parameter e. The complete recipe itself can only be obtained

upon tying the recursive knot. In the terminology of Remark 2, that is referred to as the integration. Eq_1 below integrates the appropriate clients for Φ_1, i.e., *EqNum* and *EqAdd*:

```
01  client Eq1<X ◁ Num ⊕ Add>{
02    bool equal(X x1, X x2) {
03      return (x1, x2) match {
04        case (X.Num, X.Num) ⇒ EqNum<X as Num>.eq(x1, x2, equal);
05        case (X.Add, X.Add) ⇒ EqAdd<X as Add>.eq(x1, x2, equal);
06        case _ ⇒ EqDef<X as ϵ>.eq(x1, x2, equal);
07      }
08    }
09  }
```

Likewise, one implements Eq_4 to integrate the appropriate clients for Φ_4, i.e., *EqNum*, *EqAdd*, and *EqSub*. Armed with all that, then, one can perform the following tests:

```
01  val tpfive1 = ... //3 + 5 for Φ1
02  val tpfour1 = ... //3 + 4 for Φ1
03  Eq1<Φ1>.equal(tpfive1, tpfour1) //Returns ⊥.
04  Eq4<Φ4>.equal(tpf4, tpfmo4) //Returns ⊥.
```

Observe how the technique caters for extensibility by leaving open the possibility of mixing-in new equality clients without the need to touch the existing equality cases. Note that, in this very case, the structural exercise happened to come with a default case. As noticed first by Zenger and Odersky [36] and several others afterwards, a default is not necessarily available. Section 5.2 contains an example where our solution works in the absence of default cases.

5.2. Conversion and Narrowing

Following Oliveira [22], we say an expression is **narrow**ed when all its ADT cases of a given group are cancelled into other case combinations that are deemed to be equivalent. It is common in the PL design community to provide extensions to a core PL such that the extension programs would then be narrowed to the core (for evaluation and the like). For example, GPH [43] and Utrecht Haskell [44] are both developed like that. Oliveira shows how his MVCs can be leveraged in favour of correctness for narrowing as a static guarantee that the result of this process will not contain instances from the unwanted ADT cases; it will instead contain other case combinations that are deemed equivalent.

In this section, we generalise the narrowing exercise to conversion from one ADT to another. In particular, we will craft a conversion $[\![.]\!]$ from an ADT with subtraction to an ADT with addition and negation (of signed integers): $[\![e_1 - e_2]\!] = [\![e_1]\!] + (-([\![e_2]\!]))$. To that end, we first add a new component for negation.

```
01  component Neg<X ◁ Num ⊕ Neg> {
02    X inner;
03    Neg(X inner){this.inner = inner;}
04  }
```

Then, we implement the decentralised pattern matching clients: $N2N\langle X_1 \lhd Num, X_2 \lhd Num\rangle$ to convert *Num* to *Num*, $G2G\langle X_1 \lhd Num \oplus Neg, X_2 \lhd Num \oplus Neg\rangle$ to convert *Neg* to *Neg*, and $A2A\langle X_1 \lhd Add, X_2 \lhd Add\rangle$ to convert *Add* to *Add*. The only

non-identical conversion is that of subtraction expressions to the equivalent combination of addition and negation:

```
01  client S2AN<X1 ◁ Num ⊕ Sub, X2 ◁ Num ⊕ Add ⊕ Neg> {
02    X2 convert(X1.Sub x1, X2 c(X1)) {
03      return new Add<X2>(c(x1.left), new Neg<X2>(c(x1.right)));
04    }
05  }
```

In the snippet above, X_1 and X_2 are the family parameters corresponding to the first and the second ADT, respectively. Furthermore, the parameter c of *convert* in line 02 is the function that devises the complete conversion recipe. The integration is not largely different from the ones seen thus far:

```
01  client ConvSub<X1 ◁ Num ⊕ Add ⊕ Sub ⊕ Neg, X2 ◁ Num ⊕ Add ⊕ Neg> {
02    X2 doit(X1 x1) {
03      return x1 match {
04        case X1.Num ⇒ N2N<X1 as Num, X2 as Num>.convert(x1, doit);
05        case X1.Neg ⇒ G2G<X1 as Num ⊕ Neg,
↪                          X2 as Num ⊕ Neg>.convert(x1, doit);
06        case X1.Add ⇒ A2A<X1 as Add, X2 as Add>.convert(x1, doit);
07        case X1.Sub ⇒ S2AN<X1 as Num ⊕ Sub,
↪                           X2 as Num ⊕ Add ⊕ Neg>.convert(x1, doit);
08      }
09    }
10  }
```

Narrowing would, then, be a simple application of the above conversion from an ADT to the appropriate projection of the same ADT:

```
01  client NarrowSub<X ◁ Num ⊕ Add ⊕ Sub ⊕ Neg> {
02    X doit(X x) {
03      return ConvSub<X, X as Num ⊕ Add ⊕ Neg>.doit(x);
04    }
05  }
```

Supposing the availability of the two families below

```
01  family Φ6 = Num ⊕ Add ⊕ Sub ⊕ Neg;
02  family Φ7 = Num ⊕ Add ⊕ Neg;
```

finally, one can perform the following tests:

```
val tpfmone6 = ... //(3 + 5) − 1 for Φ6
ConvSub<Φ6, Φ7>.doit(tpfmone6) //Returns (3 + 5) + (−(1)) for Φ7.
NarrowSub<Φ6>.doit(tpfmone6) //Returns (3 + 5) + (−(1)) for Φ6.
```

Remark 6. As it turns out, using the same technique, one can come up with a *widening* function which acts in the opposite direction of narrow. The interesting consequence is that, combining widen and narrow gives rise to a manual simulation for the bidirectional adaptation of $J\&_s$ [14].

Remark 7. Armed with the full arsenal of language features available in Scala, our

implementation also had the pleasure of using multiple inheritance. Hence, as shown by Haeri [26, §9.2], one also gets to a *sequential composition* [20, §17.3] variation of integration of a decentralised pattern matching. Interestingly enough, the latter variation mixes CBSE with Feature-Oriented Programming [45]. More precisely, it constitutes a verified software product-line [46, 47].

Remark 8. The straightforwardness in addressing EFP is gained by the peculiar combination of the concerns that ECP articulates. Below is more elaboration on the most remarkable points. In fact, EFP does already express its concern for a dialect of composition completeness (i.e., EC4). ECP, on the other hand, adds EC5 on top of that for sound ab initio composition and EC8 for sound extension of a composition. Our impression is that, only in the presence of both the completeness **and** soundness the integration of functions (on ADTs) is a matter of such a simple (yet strongly type-safe) relay. Without systematic handling of soundness, the programmer is on their own to manually relieve relevant anxieties of the compiler. The resulting circumvention will, unfortunately, intervene the program logic in ways that risk readability and correctness of implementation. The reader is invited to compare our online Scala implementation of the two EFP exercises with their MVC counterparts [22]. Our understanding is that the former implementation is much more readable and easier for a human-being to verify the correctness of. On the other hand, the scalability concern of ECP – i.e., EC3 – entails the ease of decentralisation. By coincidence, EC3, EC5, and EC8 are the only ECP concerns that MVCs do not score.

6. Implementation

Whilst $\gamma\Phi C_0$ still has no dedicated implementation (say a stand-alone compiler and an IDE), an embedding of its is indeed possible in Scala. Details of how to set up a complete Scala codebase for this purpose can be found in our earlier works: [48] and [26, §7 and §8]. In this section, we only explain what Scala code corresponds to what $\gamma\Phi C_0$ element of Section 2. We also study different aspects of each correspondence.

The presentations in this paper are all in Scala. However, the same solutions can be used in any host PL providing multiple inheritance and type constraints.[7] We use multiple inheritance for our cases to state their syntactic categories in addition to the ADT they are a case of. Type constraints are used as our means for checking soundness of component combinations. The Scala materials that we use here are those initially presented by Odersky and Zenger [49]. Our implementation, however, is rather inspired by *Lightweight Modular Staging* [39]. From another point of view, the implementation can also be regarded as a simulation of Polymorphic Variants [38] in Scala.

We now start by the correspondent of a $\gamma\Phi C_0$ component, which is an ordinary Scala class with specially-wired type parameterisation. Consider the class `Sub` below for the *Sub* component of Section 2:

```
1 class Sub[
2   E <: IAE[E], N <: Num[E, N] with E, S <: Sub[E, N, S] with E
3 ](left: E, right: E)
```

[7] We avoid Scala-specific terminology to make our work more broadly accessible.

Recall that the family parameterisation of *Sub* was $X \lhd Num \oplus Sub$. The translation of that is what comes in line 2 above. In their order of appearance: `E` represents X, `N` represents $X.Num$, and `S` represents $X.Sub$. In Scala, the uses of `<:` in line 2 above demand nominal subtyping. (More on `IAE` later.) Part of the verbosity in the type annotations in line 2 above is because of the idiosyncrasies of JVM that dictate similar F-Boundings [50] to the Scala type system in return of correctness.[8]

```
1  trait Phi4 extends IAE[Phi4]
2  case class Num4(n: Int) extends Num[Phi4, Num4](n) with Phi4
3  case class Add4(left: Phi4, right: Phi4) extends
4    Add[Phi4, Add4](left, right) with Phi4
5  case class Sub4(left: Phi4, right: Phi4) extends
6    Sub[Phi4, Num4, Sub4](left, right) with Phi4
```

The Scala counterpart of the Φ_4 definition is even more verbose. The trait `Phi4` in line 1 above is like the "family Φ_4" part of the family definition. `IAE` is our interface for integer arithmetic expressions. An ADT that is to use the arithmetic expression cases needs to tie the F-Bound knot of `IAE` in a similar fashion. Line 2, lines 3 and 4, and lines 5 and 6 above add the *Num*, *Add*, and *Sub* cases to `Phi4`, respectively. Notice how, unlike $\gamma\Phi C_0$ in which Φ_4 simply bundles its appropriate components together, in Scala, one cannot reuse the exact same component names for the cases of `Phi4`. For instance, the code above calls the *Add* case `Add`**4** (as opposed to just `Add`).

```
1  class ENAS[
2    E <: IAE[E],
3    N <: Num[E, N] with E,
4    A <: Add[E, A] with E,
5    S <: Sub[E, N, S] with E
6  ] {
7    def eval(x: E, e: E => Int): Int = x match {
8      case _: Add[_, _]    => new ENA[E, N, A].eval(x, e)
9      case _: Sub[_, _, _] => new ENS[E, N, S].eval(x, e)
10     case _: Num[_, _]    => new EN[E, N].eval(x, e)
11   }
12 }
```

The code above is the Scala counterpart for *ENAS* of Section 2. For the first difference, note the "`_:`" at the beginning of each pattern. In Scala, in addition to their types, patterns need names as well.

```
1  object test2 {
2    val three_plus_five4 = Add4(Num4(3), Num4(5))
3    def run1 = new Eval4[Phi4, Num4, Add4, Sub4].do_it(three_plus_five4)
4  }
```

Finally, `three_plus_five4` above constructs a corresponding expression for $3 + 5$ in Φ_4. Note the interesting advantage gained by the tight binding of `Add4` to `Phi4`: Unlike tpf_4 that requires explicit mention of Φ_4 upon every component instantiation,

[8] Alike F-Bound treatments are, in fact, popular for solving similar problems in earlier research. Consider Torgersen[8] for EP, Kamina and Tamai [11] for lightweight family polymorphism, and Bruce et al. [51] and Madsen and Ernst [28] for virtual classes.

explicit mention of `Phi4` is absent in `Add4` and `Num4` uses in line 2 above. On the contrary, because `Phi4` has no knowledge about its cases, in line 3 above, one needs to redundantly list the cases of `Phi4` upon instantiation of `Eval4` for it. Contrast that to $\gamma\Phi C_0$ version where the sole mention of Φ_4 is enough.

Remark 9. Our definition of syntactic compatibility is rather *extensional* in that it works for both structural and nominal subtyping. Here we present our implementation for nominal subtyping. However, the same techniques are applicable for structural subtyping too – especially, in a host PL like Scala with support for both sorts of subtyping [52].

7. Conclusion and Future Work

In this paper, we introduce ECP by precisely formulating its eight concerns (Section 1). We show the benefits of solving ECP along with a technology used to solve it (Section 2). We present the syntax (Section 4.1), static semantics (Appendix A), and dynamic semantics (Appendix B) of $\gamma\Phi C_0$ as a formal solution to ECP. We detail the selected parts of the $\gamma\Phi C_0$ semantics that play key roles in solving ECP (Section 4.2). Furthermore, we show how $\gamma\Phi C_0$ can easily be used for a solution to EFP (Section 5). Finally, we explain how to simulate $\gamma\Phi C_0$ in Scala (Section 6).

The immediate future work to this paper is proving the standard theoretical results (e.g., subject reduction, progress, and type soundness) about $\gamma\Phi C_0$. As stated in the introduction, ECP is a variation to EP. As such, $\gamma\Phi C_0$ is the first PL with a formal semantics that is designed for solving EP. It is especially tempting to try formal proofs for that claim. An extended future work would, then, be flourishing Section 4.2 to formal proofs for $\gamma\Phi C_0$ solving ECP. A fresh look into the paper reveals that it also introduces a new variation on family polymorphism that we would call *component family polymorphism*. Implementing $\gamma\Phi C_0$ is the other obvious future work of this paper. That would, on its own, form an interesting test bed for component family polymorphism.

Acknowldgements

This research was funded by the German Research Council (DFG). The authors would like to thank Prof. Klaus Ostermann for his comprehensive feedback that helped us upgrade from earlier versions of this paper to the current one.

References

[1] W. R. Cook, Object-Oriented Programming Versus Abstract Data Types, in: J. W. de Bakker, W. P. de Roever, G. Rozenberg (Eds.), Proc. Int. W. Found. Obj.-Oriented Lang., Vol. 489 of Lect. Notes in Comp. Sci., Noordwijkerhout (The Netherlands), 1990, pp. 151–178.

[2] J. C. Reynolds, User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction, in: S. A. Schuman (Ed.), New Dir. in Algo. Lang., 1975, pp. 157–168.

[3] P. Wadler, The Expression Problem, Java Genericity Mailing List (Nov. 1998).

[4] P. Bahr, T. Hvitved, Parametric Compositional Data Types, in: J. Chapman, P. B. Levy (Eds.), Proc. 4^{th} W. Math. Struct. Funct. Prog., Vol. 76 of Elec. Proc. Theo. Comp. Sci., 2012, pp. 3–24.

[5] M. Odersky, M. Zenger, Independently Extensible Solutions to the Expression Problem, in: Proc. Int. W. Found. Obj.-Oriented Lang., 2005.

[6] B. C. d. S. Oliveira, W. R. Cook, Extensibility for the Masses – Practical Extensibility with Object Algebras, in: Proc. 26^{th} Euro. Conf. Obj.-Oriented Progr., Vol. 7313 of Lect. Notes in Comp. Sci., Springer, 2012, pp. 2–27.
[7] W. Swierstra, Data Types à la Carte, J. Func. Prog. 18 (4) (2008) 423–436.
[8] M. Torgersen, The Expression Problem Revisited, in: M. Odersky (Ed.), Proc. 18^{th} Euro. Conf. Obj.-Oriented Progr., Vol. 3086 of Lect. Notes in Comp. Sci., Oslo (Norway), 2004, pp. 123–143.
[9] E. Ernst, Family Polymorphism, in: J. Lindskov Knudsen (Ed.), Proc. 15^{th} Euro. Conf. Obj.-Oriented Progr., Vol. 2072 of Lect. Notes in Comp. Sci., Springer, 2001, pp. 303–326.
[10] C. Saito, A. Igarashi, M. Viroli, Lightweight Family Polymorphism, J. Func. Prog. 18 (3) (2008) 285–331.
[11] T. Kamina, T. Tamai, A Design and Implementation of Lightweight Constructs for Mutually Extensible Components, Submitted Jan. 2008 to Elsevier.
[12] T. Kamina, T. Tamai, Lightweight Dependent Classes, in: Y. Smaragdakis, J. G. Siek (Eds.), Proc. Int. 7^{th} Conf. Gener. Prog. & Component Eng., ACM, Nashville, TN, USA, 2008, pp. 113–124.
[13] N. Nystrom, X. Qi, A. C. Myers, J&: Nested Intersection for Scalable Software Composition, in: Proc. 21^{st} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., ACM, Portland, Oregon, USA, 2006, pp. 21–36.
[14] X. Qi, A. C. Myers, Sharing Classes between Families, in: M. Hind, A. Diwan (Eds.), Proc. ACM SIGPLAN Conf. Prog. Lang. Design & Impl., ACM, 2009, pp. 281–292.
[15] X. Qi, A. C. Myers, Homogeneous Family Sharing, in: W. R. Cook, S. Clarke, M. C. Rinard (Eds.), Proc. 25^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., ACM, Reno/Tahoe, Nevada, USA, 2010, pp. 520–538.
[16] T. Budd, An Introduction to Object-Oriented Programming, 2^{nd} Edition, Addison-Wesley, 1997.
[17] J. Heering, P. Klint, The Prehistory of ASF+SDF (1980–1984), in: M. G. J. v. d. Brand, A. v. Deursen, T. B. Dinesh, J. Kamperman, E. Visser (Eds.), Proc. ASF+SDF'95 W. Gener. Tools Alg. Spec., Technical Report P9504, Programming Research Group, University of Amsterdam, 1995, pp. 1–4.
[18] R. M. Burstall, J. A. Goguen, The Semantics of Clear, a Specification Language, in: Abstract Software Specification, Copenhagen Winter School, Vol. 86 of Lect. Notes in Comp. Sci., Springer, NY, USA, 1980, pp. 292–332.
[19] M. Cerioli, J. Meseguer, May I Borrow Your Logic? (Transporting Logical Structures along Maps), Theo. Comp. Sci. 173 (1997) 311–347.
[20] I. Sommerville, Software Engineering, 9^{th} Edition, Addison Wesley, 2011.
[21] R. S. Pressman, Software Engineering: A Practitioner's Approach, 7^{th} Edition, McGraw-Hill, 2009.
[22] B. C. d. S. Oliveira, Modular Visitor Components, in: Proc. 23^{rd} Euro. Conf. Obj.-Oriented Progr., Vol. 5653 of Lect. Notes in Comp. Sci., Springer, 2009, pp. 269–293.
[23] C. Szyperski, Independently Extensible Systems – Software Engineering Potential and Challenges, in: Proc. 19^{th} Australasian Comp. Sci. Conf., 1996.
[24] S. Erdweg, P. G. Giarrusso, T. Rendel, Language Composition Untangled, in: Proc. 5^{th} Int. W. Lang. Descr. Tools & App., ACM, 2012, pp. 49–56.
[25] C. Hofer, K. Ostermann, Modular Domain-Specific Language Components in Scala, in: E. Visser, J. Järvi (Eds.), Proc. Int. 9^{th} Conf. Gener. Prog. & Component Eng., ACM, Eindhoven, The Netherlands, 2010, pp. 83–92.
[26] S. H. Haeri, Component-Based Mechanisation of Programming Languages in Embedded Settings, Ph.D. thesis, Institute for Software Systems, Hamburg University of Technology, Hamburg, Germany, submitted (Jul. 2014).
[27] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: A Minimal Core Calculus for Java and GJ, Trans. Prog. Lang. & Sys. 23 (3) (2001) 396–450.
[28] A. B. Madsen, E. Ernst, Revisiting Parametric Types and Virtual Classes, in: J. Vitek (Ed.), Obj., Models, Components, Patterns, 48^{th} Int. Conf. Proc., Vol. 6141 of Lect. Notes in Comp. Sci., Springer, 2010, pp. 233–252.
[29] E. Ernst, gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance, Ph.D. thesis, Dept. Comp. Sci. , Uni. Århus, Århus, Denmark (1999).
[30] E. Ernst, Reconciling Virtual Classes with Genericity, in: D. E. Lightfoot, C. A. Szyperski (Eds.), Proc. Modular Prog. Lang., 7^{th} Joint Modular Lang. Conf., Vol. 4228 of Lect. Notes in Comp. Sci., Springer, 2006, pp. 57–72.
[31] N. Nystrom, S. Chong, A. C. Myers, Scalable Extensibility via Nested Inheritance, in: J. M. Vlissides, D. C. Schmidt (Eds.), Proc. 19^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., ACM, 2004, pp. 99–115.

[32] A. B. Compagnoni, B. C. Pierce, Higher-Order Intersection Types and Multiple Inheritance, Math. Struct. Com. Sci. 6 (5) (1996) 469–501.
[33] P. Jolly, S. Drossopoulou, C. Anderson, K. Ostermann, Simple Dependent Types: Concord, in: Proc. 6^{th} W. Formal Tech. Java-like Prog., 2004, Tech. Rep. nr. NIII-R0426, Uni. Nijmegen.
[34] Guttag, J. V. and Horning, J. J., The Algebraic Specification of Abstract Data Types, Acta Informatica 10 (1978) 27–52.
[35] B. C. d. S. Oliveira, T. van der Storm, A. Loh, W. R. Cook, Feature-Oriented Programming with Object Algebras, in: G. Castagna (Ed.), Proc. 27^{th} Euro. Conf. Obj.-Oriented Progr., Vol. 7920 of Lect. Notes in Comp. Sci., Springer, Montpellier, France, 2013, pp. 27–51.
[36] M. Zenger, M. Odersky, Extensible Algebraic Datatypes with Defaults, in: Proc. 6^{th} ACM SIGPLAN Int. Conf. Func. Prog., ACM, Firenze (Florence), Italy, 2001, pp. 241–252.
[37] J. Garrigue, Code Reuse through Polymorphic Variants, in: W. Found. Soft. Eng., no. 25, 2000, pp. 93–100.
[38] J. Garrigue, Programming with Polymorphic Variants, in: X. Leroy, D. MacQueen (Eds.), Proc. 5^{th} ACM SIGPLAN W. ML, Baltimore, MD, USA, 1998.
[39] T. Rompf, M. Odersky, Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs, in: Proc. Int. 9^{th} Conf. Gener. Prog. & Component Eng., ACM, Eindhoven, The Netherlands, 2010, pp. 127–136.
[40] R. C. Martin, Design Principles and Design Patterns, online article available from the ObjectMentor website (2000).
[41] C. Chambers, G. T. Leavens, Typechecking and Modules for Multimethods, Trans. Prog. Lang. & Sys. 17 (6) (1995) 805–843.
[42] C. Clifton, G. T. Leavens, C. Chambers, T. D. Millstein, MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, in: Proc. 15^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., ACM, Minneapolis, Minnesota, USA, 2000, pp. 130–145.
[43] P. Trinder, K. Hammond, H.-W. Loidl, S. Peyton Jones, Algorithm + Strategy = Parallelism, J. Func. Prog. 8 (1) (1998) 23–60.
[44] A. Dijkstra, J. Fokker, S. D. Swierstra, The Architecture of the Utrecht Haskell Compiler, in: Proc. 2^{nd} ACM SIGPLAN Symp. on Haskell, ACM, Edinburgh, Scotland, 2009, pp. 93–104.
[45] C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, in: M. Aksit, S. Matsuoka (Eds.), Proc. 11^{th} Euro. Conf. Obj.-Oriented Progr., Vol. 1241 of Lect. Notes in Comp. Sci., Jyväskylä, Finland, 1997, pp. 419–443.
[46] S. Thaker, D. S. Batory, D. Kitchin, W. R. Cook, Safe Composition of Product Lines, in: C. Consel, J. L. Lawall (Eds.), Proc. Int. 6^{th} Conf. Gener. Prog. & Component Eng., ACM, Salzburg, Austria, 2007, pp. 95–104.
[47] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A Classification and Survey of Analysis Strategies for Software Product Lines, ACM Computing Surveys 47 (1) (2014) 6:1–6:45.
[48] S. H. Haeri, S. Schupp, Reusable Components for Lightweight Mechanisation of Programming Languages, in: W. Binder, E. Bodden, W. Löwe (Eds.), Proc. 12^{th} Int. Conf. Soft. Composition, Vol. 8088 of Lect. Notes in Comp. Sci., Springer, 2013, pp. 1–16.
[49] M. Odersky, M. Zenger, Scalable Component Abstractions, in: Proc. 20^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., ACM, San Diego, CA, USA, 2005, pp. 41–57.
[50] P. Canning, W. Cook, W. Hill, W. Olthoff, J. C. Mitchell, F-Bounded Polymorphism for Object-Oriented Programming, in: FPCA '89: Proc. 4^{th} Int. Conf. Func. Prog. Lang. & Comp. Arch., 1989, pp. 273–280.
[51] K. B. Bruce, M. Odersky, P. Wadler, A Statically Safe Alternative to Virtual Types, in: E. Jul (Ed.), Proc. 12^{th} Euro. Conf. Obj.-Oriented Progr., Vol. 1445 of Lect. Notes in Comp. Sci., Springer, 1998, pp. 523–549.
[52] G. Dubochet, M. Odersky, Compiling Structural Types on the JVM: a Comparison of Reflective and Generative Techniques from Scala's Perspective, in: Proc. 4^{th} W. Impl., Comp., Opt. of Obj.-Oriented Lang. & Prog. Sys., ACM, Genova, Italy, 2009, pp. 34–41.
[53] B. Liskov, Keynote Address – Data Abstraction and Hierarchy, ACM SIGPLAN Not. 23 (5) (1987) 17–34.

Appendix A. The Static Semantics of $\gamma\Phi C_0$

This section aims at providing typing rules for $\gamma\Phi C_0$, as done in Appendix A.6. The major tool to that end are well-formedness (Appendix A.4). The other subsections

provide other utilities for that purpose: Appendix A.1 presents the lookup functions; Appendix A.2 discusses the subtyping rules; and, Appendix A.3 and Appendix A.5 offer other utilities for well-formedness and typing respectively.

Given the list comprehension notation that we borrow from the Lightweight Family Polymorphism literature (Section 3), the rules of our static and dynamic semantics are likely to take time to absorb. Here are a few rules of thumb that will accelerate the reader: Whenever, in a phrase, a list of items is used in a place where a single item of the type is expected, the comprehension is abbreviating a list of phrases rather than a single one. For example, the phrase $G \vdash \overline{R}$ ok denotes the following list of judgements $G \vdash R_1$ ok, $\ldots, G \vdash R_n$ ok, for the appropriate n. When, in a phrase, there are two comprehensions in two places where a single item is expected, a list of phrases is meant, where the two lists are iterated together. For instance, $\overline{\gamma}\!<\!Z\!>\!/X.\overline{\gamma}$ denotes $\gamma_1\!<\!Z\!>\!/X.\gamma_1, \ldots, \gamma_n\!<\!Z\!>\!/X.\gamma_n$. Note that, in every substitution, the γ on the left-hand-side is of the **same-index** that the right-hand-side one is of. As stated earlier on too, in our semantics, we refer to that property as the **same-indexed** convention.

We maintain a similar convention for iteration of three lists together for when three single items are expected in a phrase. For example, $\mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{v} : \overline{A'}$ abbreviates $\mathit{as\text{-}fam}(Z_1) \vdash v_1 : A'_1, \ldots, \mathit{as\text{-}fam}(Z_n) \vdash v_n : A'_n$, where $\overline{Z}$, $\overline{v}$, and $\overline{A'}$ are iterated together to obtain the individual judgements. On the contrary, when, in a phrase, there are four comprehensions in places where four single items are expected, there is a nesting of iteration loops, in each loop a pair of comprehensions are to be iterated together. For example, for $\overline{R}[Z/X, \overline{\gamma}\!<\!Z\!>\!/X.\overline{\gamma}]\ \overline{f}$, the pair $\overline{R}$ and $\overline{f}$ are iterated together, and the pair $\overline{\gamma}\!<\!Z\!>$ and $X.\overline{\gamma}$ are iterated together (inside the first iteration). Throughout the appendices, whenever such unfolding actions are needed, we explain the instances.

Appendix A.1. Lookup Functions

Before we move to the lookup functions themselves, we explain some auxiliaries functions that will become handy over the lookup. Define 'the *component mix of a selection*' $\mathit{cmixs}(\Phi) = (\Phi, \oplus\overline{\gamma})$ when $\Phi = \oplus\overline{\gamma}$ and $\mathit{cmixs}(\Phi\ \mathsf{as}\ \oplus\overline{\gamma}) = (\Phi, \oplus\overline{\gamma})$. As another auxiliary, define $\mathit{fps}(G\!<\!X_1 \lhd \ldots, X_2 \lhd \ldots, \ldots, X_n \lhd \ldots\!>) = X_1, X_2, \ldots, X_n$. On top of those two, define $\mathit{pfcmix}(G\!<\!\overline{Z}\!>, X_i) = \mathit{cmixs}(Z_i)$ when either $X_i \in \mathit{fps}(G)$. In words, $\mathit{pfcmix}(I, X)$ is 'the *passed family and components mixture* for parameter X to get I.' Finally, define 'the *family parameter upper bound*' $\mathit{fpub}(G\!<\ldots, X\ \mathsf{as}\ \oplus\overline{\gamma}, \ldots\!>, X) = \oplus\overline{\gamma}$.

$$\mathit{fields}(X.\gamma) = \mathit{fields}(\gamma) \qquad \text{(F-VCase)} \qquad\qquad \mathit{fields}(X) = \epsilon \qquad \text{(F-FPar)}$$

$$\frac{\mathsf{component}\ \gamma\!<\cdots\!> \lhd \cdots \{\overline{Rf}\,;\ \cdots\}}{\mathit{fields}(\gamma) = \overline{Rf}}\ \text{(F-Comp)}$$

$$\mathit{fields}(\gamma\!<\!X\ \mathsf{as}\ \oplus\overline{\gamma}\!>) = \mathit{fields}(\gamma) \qquad \text{(F-FProj)}$$

$$\frac{\mathit{cmixs}(Z) = (\Phi, \oplus\overline{\gamma}) \qquad \gamma \in \oplus\overline{\gamma} \qquad \mathit{fields}(\gamma) = \overline{R}\,\overline{f}}{\mathit{fields}(\gamma\!<\!Z\!>) = \overline{R}[Z/X, \overline{\gamma}\!<\!Z\!>\!/X.\overline{\gamma}]\ \overline{f}}\ \text{(F-Case)}$$

Figure A.5: Field Lookup of $\gamma\Phi C_0$

Figures A.5 and A.6 are routine lookup functions for the fields and method types. We start the explanation by the former. The rule (F-VCase) is an immediate implication of Remark 10. Recall that $X.\gamma \overset{s}{\equiv} \gamma$. A family parameter has no fields, hence, (F-FPar). The rule (F-Comp) states that the fields of a component are those mentioned in its own body. (Note that we maintain a convention on the field names being distinct.)

It is, finally, when a family (or a projection of it) is used for component instantiation that relative types are replaced by absolute ones. (F-Case) formulates the process for fields. In the last rule, the abbreviation is doubly-folded: For a fixed X and a fixed Z, the phrase $\overline{\gamma}\!<\!Z\!>\!/X.\overline{\gamma}$ stands for $\gamma_1\!<\!Z\!>\!/X.\gamma_1, \ldots, \gamma_n\!<\!Z\!>\!/X.\gamma_n$. Atop, $\overline{R}[\cdots]\ \overline{f}$ abbreviates $R_1[\cdots]\ f_1, \ldots, R_n[\cdots]\ f_n$. (Namely, inside the pair of square bracket lies one folding; there is another folding outside the pair.) Note that the replacement process is order-sensitive. That is, for every appropriate i, the relative $X.\gamma_i$ is replaced by the **same-indexed** component instance $\gamma_i\!<\!\Phi\!>$. Likewise, for every appropriate j, the relative type R_j is that of the field f_j.

$$\frac{G\{\cdots \overline{M} \cdots\} \qquad R\ m(\overline{R}\ \overline{x})\{\cdots\} \in \overline{M}}{\mathit{mtype}(m, G) = \overline{R} \to R}\ \text{(MT-G)}$$

$$\frac{\begin{array}{c} I = C\!<\!\cdots\!> \qquad \mathit{fps}(C) = \overline{X} \qquad \mathit{mtype}(m_C, C) = \overline{R} \to R \\ \mathit{fpub}(C, \overline{X}) = \overline{\oplus\overline{\gamma}'} \qquad \mathit{pfcmix}(I, \overline{X}) = (\overline{\Phi}, \overline{\oplus\overline{\gamma}}) \end{array}}{\mathit{mtype}(m_C, I) = (\overline{R} \to R)[\overline{\Phi}/\overline{X}, \overline{\gamma}\!<\!\overline{\Phi}\!>\!/\overline{X}.\overline{\gamma}']}\ \text{(MT-Inst)}$$

Figure A.6: Method Type Lookup of $\gamma\Phi C_0$

The method type of m in G, as stated by (MT-G), is trivially calculated using its nominated argument and return types. In (MT-Inst), the method type of m in an instance of a client C is that of C itself, updated with the instantiation information. The notation $\overline{\gamma}\!<\!\overline{\Phi}\!>\!/\overline{X}.\overline{\gamma}'$ is again doubly-folded and deserves dedicated unfolding advice. $\overline{\gamma}\!<\!\overline{\Phi}\!>\!/\overline{X}.\overline{\gamma}'$ stands for $\overline{\gamma}\!<\!\Phi_1\!>\!/X_1.\overline{\gamma}', \ldots, \overline{\gamma}\!<\!\Phi_n\!>\!/X_n.\overline{\gamma}'$. Besides, for a fixed X and a fixed Φ, $\overline{\gamma}\!<\!\Phi\!>\!/X.\overline{\gamma}'$ stands for $\gamma_1\!<\!\Phi\!>\!/X.\gamma_1', \ldots, \gamma_k\!<\!\Phi\!>\!/X.\gamma_k'$. That is, the **same-indexed** convention is maintained in both foldings here. We maintain the same-indexed convention throughout $\gamma\Phi C_0$. In particular, component combinations are considered in their exact same order that they are listed by the programmer. The $\gamma\Phi C_0$ semantics does not put effort into trying other (equally meaningful) permutations.

Appendix A.2. Subtyping

The subtyping rules of $\gamma\Phi C_0$ are presented in Figure A.7. In line with Figure 1, the rules are divided into those pertaining to prior family introduction (Relative Subtyping) and after that (Absolute Subtyping). In the former group, the decision is made with respect to the relative enclosing definition (G), whereas, for the latter, a family (Φ) is the basis of decision. The reflexiveness and transitivity rules are standard. The other two are specifically relevant to ECP. They are along the lines that the cases of an ADT are all different forms of it. Note that these last two rules are likely not to be correct for CBSE applied in other disciplines. In fact, whilst other disciplines are entitled to add their own rules here, it is only the reflexiveness and transitivity rules to expect from $\overset{s}{\equiv}$.

Relative Subtyping $\boxed{G \vdash R <: R'}$

$$\frac{}{G \vdash R <: R} \text{ (S-ReflG)} \qquad \frac{G \vdash R <: R' \qquad G \vdash R' <: R''}{G \vdash R <: R''} \text{ (S-TranG)}$$

$$\frac{fpub(G, X) = \oplus\overline{\gamma} \qquad \gamma \in \oplus\overline{\gamma}}{G \vdash X.\gamma <: X} \text{ (S-VCase)}$$

Absolute Subtyping $\boxed{\Phi \vdash A <: A'}$

$$\frac{}{\Phi \vdash A <: A} \text{ (S-ReflF)} \qquad \frac{\Phi \vdash A <: A' \qquad \Phi \vdash A' <: A''}{\Phi \vdash A <: A''} \text{ (S-TranF)}$$

$$\frac{\Phi = \oplus\overline{\gamma} \qquad \gamma \in \oplus\overline{\gamma}}{\Phi \vdash \gamma\text{<}\Phi\text{>} <: \Phi} \text{ (S-Case)}$$

Figure A.7: The Two Sorts of Subtyping in $\gamma\Phi C_0$

Remark 10. As announced in Section 4.1, $\gamma\Phi C_0$ assumes the availability of an equivalence relation $\overset{s}{\equiv}$. Replacement of a component by an equivalent should go unobserved by the recipient code. The two aspects of this unobservedness are: constructibility with the same syntax and provision of the same interface. Note that, in $\gamma\Phi C_0$, the latter condition also means that they must come with the same fields and similar 'requires' interfaces. The presence of those aspects makes the equivalent component substitution in *ok-at*(.) safe (Figure A.9). Note that, whilst we require equivalent components to share the observed interface, their implementation of the requested services – as well as the rest of their interfaces – can well differ.[9] Unlike the case rules of Figure A.7, we find the necessity of $\overset{s}{\equiv}$ to be a property of CBSE in general (rather than ECP exclusively).

Many real-world PLs do already have means that can be leveraged to guarantee the above unobservedness. Examples are the ordinary C++ template mechanism, C++ concepts or Haskell type classes, and the implicits of Scala. The study of advantages and disadvantages of each means is a topic for future research.

Appendix A.3. Other Well-Formedness Utilities

$$\frac{fpub(G, X) = \oplus\overline{\gamma} \qquad \{\overline{\gamma_p}\} \subseteq \{\overline{\gamma}\}}{valid\text{-}as(G, X, \oplus\overline{\gamma_p})} \text{ (Valid-As-G)}$$

$$\frac{\Phi = \oplus\overline{\gamma} \qquad \{\overline{\gamma_p}\} \subseteq \{\overline{\gamma}\}}{valid\text{-}as(\Phi, \oplus\overline{\gamma_p})} \text{ (Valid-As-F)}$$

Figure A.8: Valid Projection of Component Combinations in $\gamma\Phi C_0$

We stated in Section 4.2 that *valid-as*(.) is at the heart of $\gamma\Phi C_0$'s support for EC8. Figure A.8 details that function. *valid-as*(.) answers the question of whether the component combination passed as its second argument is a valid projection of its first argument.

[9] This is similar to the *substitutability* of Liskov [53], except that her work was exclusive to inheritance.

The essence of *valid-as*(.) is the simple subset check $\{\overline{\gamma_p}\} \subseteq \{\overline{\gamma}\}$ where $\oplus\overline{\gamma}$ is the available component combination and $\oplus\overline{\gamma_p}$ is the requested projection.[10] The rule (VALID-AS-G) conducts the validity check for components and clients, whilst (VALID-AS-F) does that for families. We now continue to a series of other auxiliaries.

Define $fpub^*(.)$ as the transitive closure of $fpub(.)$:

$$fpub^*(\gamma) = fpub(\gamma) \cup \bigcup_{\gamma' \in fpub(\gamma)} fpub^*(\gamma').$$

Define also the set of *unwanted* components for a component γ as $ucomps(\mathsf{component}\ \gamma\!<\cdots \not\lhd \overline{\gamma}\!>) = \overline{\gamma}$. Next, define 'the *passed family selection* to an *instance*' $psfi(C\!<\!\overline{Z}\!>, X_i) = Z_i$ when $X_i \in fps(C)$. On top of the definitions of this section, we define more complicated utilities in Figure A.9. In words, the rule (NC-COMP) manifests it that so long as the transitive closure of the components in the 'requires' interface of a given set of components $\overline{\gamma}$ has no intersection with the components that are unwanted by $\overline{\gamma}$, they are in no conflict. Using that rule, then, the rule (NC-CLI) explains that a client has no conflict at its family parameter X when the upper bound of X is non-conflicting.

The rules (OA-CFINST) and (OA-CPINST) are both on legislation of substituting family selections for family parameters of a client. They consider the cases where a full family or a projection of it is used for that purpose, respectively. The essence of the two is their last two premises: $\#\overline{\gamma} = \#\overline{\gamma}'$ and $\overline{\gamma} \overset{s}{\equiv} \overline{\gamma}'$. The first condition checks whether the passed combination does indeed have the same length of the requested combination. Then, the second checks whether every passed component is in $\overset{s}{\equiv}$ relationship with its **same-indexed** requested one.

$$\frac{fpub^*(\overline{\gamma}) \cap \bigcap_{\gamma \in fpub^*(\overline{\gamma})} ucomps(\gamma) = \varnothing}{non\text{-}conf(\overline{\gamma})}\ (\text{NC-COMP})$$

$$\frac{fpub(C, X) = \oplus\overline{\gamma} \qquad non\text{-}conf(\overline{\gamma})}{non\text{-}conf(C, X)}\ (\text{NC-CLI})$$

$$\frac{\begin{array}{lll} I = C\!<\cdots> & psfi(I, X) = \Phi & \Phi = \oplus\overline{\gamma} \\ fpub(C, X) = \oplus\overline{\gamma}' & \#\overline{\gamma} = \#\overline{\gamma}' & \overline{\gamma} \overset{s}{\equiv} \overline{\gamma}' \end{array}}{ok\text{-}at(I, X)}\ (\text{OA-CFINST})$$

$$\frac{\begin{array}{lll} I = C\!<\cdots> & psfi(I, X) = \Phi\ \mathsf{as}\ \overline{\gamma} & valid\text{-}as(\Phi, \overline{\gamma}) \\ fpub(C, X) = \oplus\overline{\gamma}' & \#\overline{\gamma} = \#\overline{\gamma}' & \overline{\gamma} \overset{s}{\equiv} \overline{\gamma}' \end{array}}{ok\text{-}at(I, X)}\ (\text{OA-CPINST})$$

Figure A.9: *non-conf*(.) and *ok-at*(.)

[10] The "p" subscript in γ_p is for projection.

Appendix A.4. Well-Formedness

In correspondence with Figure 1, Figures A.10 and A.11 divide the well-formedness rules into two parts: relative and absolute, respectively. When a relative entity r is well-formed in G, we write "$G \vdash r$ ok". Likewise, when an absolute entity a is well-formed, we write "a ok". The rest of this section provides informal explanation for both.

$$\frac{\overline{x} : \overline{R}, \mathsf{this} : X.\gamma; \gamma \vdash e_\gamma : R' \qquad \gamma \vdash R' <: R \qquad \gamma \vdash R, R', \overline{R}\ \text{ok}}{\gamma \vdash R\ m_\gamma(\overline{R}\ \overline{x})\{\mathsf{return}\ e_\gamma\mathsf{;}\}\ \text{ok}}\ (\text{WF-MC}\textsc{omp})$$

$$\frac{\overline{x} : \overline{R}; C \vdash e_C : R' \qquad C \vdash R' <: R \qquad C \vdash R, R', \overline{R}\ \text{ok}}{C \vdash R\ m_C(\overline{R}\ \overline{x})\{\mathsf{return}\ e_C\mathsf{;}\}\ \text{ok}}\ (\text{WF-MC}\textsc{li})$$

$$\frac{\mathit{fields}(\gamma) = \overline{R}\ \overline{f}}{\gamma(\overline{R}\ \overline{f})\{\mathsf{this}.\overline{f} = \overline{f}\mathsf{;}\}\ \text{ok}}\ (\text{WF-C}\textsc{tor})$$

$$\frac{\mathit{non\text{-}conf}(\gamma) \qquad K\ \text{ok} \qquad \gamma \vdash \overline{M_\gamma}\ \text{ok} \qquad \gamma \vdash \overline{R}\ \text{ok}}{\mathsf{component}\ \gamma\{\overline{R}\ \overline{f}\mathsf{;}\ K\ \overline{M_\gamma}\}\ \text{ok}}\ (\text{WF-C}\textsc{omp})$$

$$\frac{\overline{X} = \mathit{fps}(C) \qquad \mathit{non\text{-}conf}(C, \overline{X}) \qquad C \vdash \overline{M_C}\ \text{ok}}{\mathsf{client}\ C\{\overline{M_C}\}\ \text{ok}}\ (\text{WF-C}\textsc{li})$$

$$\frac{}{G \vdash \epsilon\ \text{ok}}\ (\text{WF-E}\textsc{mpty}) \qquad \frac{X \in \mathit{fps}(G)}{G \vdash X\ \text{ok}}\ (\text{WF-FP}\textsc{ar})$$

$$\frac{\mathit{fpub}(G, X) = \oplus\overline{\gamma} \qquad \gamma \in \oplus\overline{\gamma}}{G \vdash X.\gamma\ \text{ok}}\ (\text{WF-VC}\textsc{ase})$$

Figure A.10: Relative Well-Formedness in $\gamma\Phi C_0$

The rules (WF-MC$\textsc{omp}$) and (WF-MC$\textsc{li}$) are routine. They pertain to the well-formedness of component and client methods, respectively. Note that $G \vdash \overline{R}$ ok abbreviates $G \vdash R_1$ ok, $\ldots, G \vdash R_n$ ok. (WF-C$\textsc{tor}$) is about (component) constructors and is routine. (WF-C$\textsc{omp}$) states that a component is well-formed when there is no conflict between its (transitively) requested and unwanted components and the following are all well-formed: its constructor, its methods, and field types. Given that a client has no constructor or fields, (WF-C$\textsc{li}$) is similar but only checks the well-formedness of (client) methods. The other difference is that (WF-C$\textsc{li}$) requires the absence of conflict separately for each (and every) of its family parameters. Note that $\mathit{non\text{-}conf}(C, \overline{X})$ abbreviates $\mathit{non\text{-}conf}(C, X_1), \ldots, \mathit{non\text{-}conf}(C, X_n)$. According to (WF-E$\textsc{mpty}$), the empty list is always well-formed w.r.t. a relative definition. (WF-FP$\textsc{ar}$) states that the family variables of a relative definition are all well-defined w.r.t. to it. Finally, due to (WF-VC$\textsc{ase}$), $X.\gamma$ – i.e., the component γ of the family parameter X – is only well-formed when the enclosing relative definition G has already enforced the availability of γ in the family to substitute for X to instantiate G.

The rule (WF-F$\textsc{amily}$) is already explained in Section 4.2. It only remains to present two formal definitions. Write $\mathit{sat\text{-}by}(\gamma, \Phi)$ when $\exists \gamma' \in \Phi.\ \gamma \overset{s}{\equiv} \gamma'$. We also overload the

$$\frac{\overline{\gamma} = \mathit{fpub}^*(\Phi) \qquad \mathit{sat\text{-}by}(\overline{\gamma}, \Phi) \qquad \mathit{non\text{-}conf}(\overline{\gamma})}{\mathsf{family}\ \Phi = \cdots\ \mathrm{ok}}\ (\text{WF-FAMILY})$$

$$\frac{I = C{<}\cdots{>} \qquad C\ \mathrm{ok} \qquad \overline{X} = \mathit{fps}(C) \qquad \mathit{ok\text{-}at}(I, \overline{X})}{I\ \mathrm{ok}}\ (\text{WF-INST})$$

Figure A.11: Well-Formedness of Absolutes in $\gamma\Phi C_0$

$\mathit{fpub}^*(.)$ function for families: $\mathit{fpub}^*(\Phi) = \bigcup_{\gamma \in \Phi} \mathit{fpub}^*(\gamma)$. The rule (WF-INST) mentions the conditions for the well-formedness of an instance. First, the instantiated client itself needs to be well-formed. Second, the family selections sent as arguments to each (and every) family parameter should be acceptable. Note that $\mathit{ok\text{-}at}(I, \overline{X})$ abbreviates $\mathit{ok\text{-}at}(I, X_1), \ldots, \mathit{ok\text{-}at}(I, X_n)$.

Appendix A.5. Typing Utilities

In this section, we define a handful of auxiliary functions that we use in the typing rules of $\gamma\Phi C_0$ (Appendix A.6). We start by the function $\mathit{fp}(S)$ for the family parameter of a selection S. Define $\mathit{fp}(X) = X$ and $\mathit{fp}(X\ \mathsf{as}\ \oplus\overline{\gamma}) = X$. Furthermore, define the '*component combination* used in a *selection* for a *client*' – denoted by $\mathit{cls\text{-}ccomb}(.)$ – as follows:

$$\begin{cases} \mathit{cls\text{-}ccomb}(C, X\ \mathsf{as}\ \oplus\overline{\gamma}) = \oplus\overline{\gamma} & \text{when} \quad X \in \mathit{fps}(C) \\ \mathit{cls\text{-}ccomb}(C, X) = \oplus\overline{\gamma} & \text{when} \quad \mathit{fpub}(C, X) = \oplus\overline{\gamma} \end{cases}$$

The last simple function that we define here is $\mathit{projs}(C, X)$ for the list of projections of X in the body of C. (We do not present a more precise definition for $\mathit{projs}(.,.)$.)

The above few functions pave the road for more complicated ones that will be of direct use in the next section. Consider Figure A.13. A predicate $C \vdash \mathit{ssfp}(S, X', C')$ ok claims that 'substitution of the family parameter selection S for the family parameter X' of C' is acceptable. The rules (S-WFP) and (S-PFP) list the conditions that justify such a predicate. They correspond to the situations when S is a family parameter X itself and when a projection of it is used.

Last comes the rule (FC), which enumerates the conditions in which the notation $R[C \xrightarrow{[S/X']} C']$ is defined in addition to what it represents. That notation is for updating a relative type R acquired from a call to a C' method for the use within a client C. When the roles of C and C' are not of interest and we are rather after the update action itself, we use the shorter notation $R[X' \mapsto S]$. The last group of substitutions advised by the rule (FC) in its right-hand-side of the conclusion deserves further explanation: The (FC) rule aside, when $\{\overline{\gamma}'\} \subseteq \{\overline{\gamma}\}$, we generally use $(X\ \mathsf{as}\ \oplus\overline{\gamma})\ \mathsf{as}\ \oplus\overline{\gamma}'$ for $X\ \mathsf{as}\ \oplus\overline{\gamma}'$.

Appendix A.6. Typing

Figure A.13 illustrates the typing rules for $\gamma\Phi C_0$ expressions. The scheme of these rules is $\Gamma; G \vdash e : R$. Here, Γ is a type environment, which is a partial function from variable names to their types. The scheme reads 'with the type environment Γ and inside G, the expression e is typed to R.'

The rules (T-VAR) and (T-FIELD) are routine. The rule (T-INVK$_1$) is less straightforward in that it allows the arguments passed to a method to also be of subtypes of the respective nominated parameter types. (T-NEW) offers the same flexibility when sending arguments to a component γ's constructor. In addition, it checks if γ is basically well-formed in G. (T-INVK$_2$) is like (T-INVK$_1$) but simpler: After all, unlike an m_γ, an m_C has no receiver expression. The rule (T-INVK$_3$) is already explained in Section 4.2.

Figure A.14 is on our single rule for typing a test τ. (Recall the τ syntax in Figure 1.) Given that this rule was already explained in Section 4.2, we drop further explanation here. It remains for us, nevertheless, to give formal definitions for the auxiliary functions used in it. The '*family selection* of a value' is defined as $\mathit{fsel}(\mathsf{new}\ \gamma{<}Z{>}(\epsilon)) = Z$ and $\mathit{fsel}(\mathsf{new}\ \gamma{<}Z{>}(\overline{v})) = Z$ if and only if $\mathit{fsel}(\overline{v}) = Z$. Moreover, define a 'family selection considered *as* a stand-alone *family*' as follows: $\mathit{as\text{-}fam}(\Phi) = \Phi$ and $\mathit{as\text{-}fam}(\Phi\ \mathsf{as}\ \oplus\overline{\gamma}) = \Phi'$ where the fresh family Φ' is defined to be $\Phi' = \oplus\overline{\gamma}$ when $\mathit{valid\text{-}as}(\Phi, \oplus\overline{\gamma})$.

Appendix B. The Dynamic Semantics of $\gamma\Phi C_0$

Although we do not really make use of the $\gamma\Phi C_0$ dynamic semantics in this paper, this section presents the dynamic semantics for completeness purposes. The $\gamma\Phi C_0$ dynamic semantics is remarkably less complicated than its static semantics. Before we get into the dynamic semantics, we would like to draw the reader's attention to the following comparison about Figure 1: Although a client expression e_C can take the form $m_C(\overline{e_C})$, there is no corresponding form for an absolute expression e_a. That is indeed a design choice we made in the favour of simplicity of the dynamic semantics. More precisely, once the static semantics of a $\gamma\Phi C_0$ program is checked, we assume any method call $m_C(\overline{e_C})$ in a client $C{<}\cdots{>}$ to have been normalised to their equivalent $C{<}\cdots{>}.m_C(\overline{e_C})$ call. As such, calls of the $m_C(\overline{e_C})$ form will no longer exist in a program to take a counterpart form in e_a.

After such normalisations, similar to the familiar type instantiation, $\gamma\Phi C_0$ offers a **family parameter substitution** phase. To get to the formal definition, we first present the function $\mathit{rtypes}(.)$ for the relative types of a relative expression:

$$
\begin{array}{ll}
\mathit{rtypes}(x) & = \epsilon \\
\mathit{rtypes}(e_r.f) & = \mathit{rtypes}(e_r) \\
\mathit{rtypes}(e_r.m_\gamma(\overline{e_r})) & = \mathit{rtypes}(e_r), \mathit{rtypes}(\overline{e_r}) \\
\mathit{rtypes}(\mathsf{new}\ X.\gamma(\overline{e_r})) & = X.\gamma, \mathit{rtypes}(\overline{e_r}) \\
\mathit{rtypes}(C{<}\overline{S}{>}.m(\overline{e_C})) & = \mathit{fps}(\overline{S}), \mathit{rtypes}(\overline{e_C})
\end{array}
$$

Next, define $\mathit{fps}(e_C) = \{X \mid X \in \mathit{rtypes}(e_C) \ \vee\ X.\_ \in \mathit{rtypes}(e_C)\}$ for the family parameters that occur in an expression e_C.

We can now move to the function $\mathit{mbody}(.,.)$ in Figure B.15. The function $\mathit{mbody}(m, G)$ returns the body of a method m of G along with the set of formal parameters that m takes and the definition site of m. We overload the $\mathit{mbody}(.,.)$ function to also handle the case when a G is to be instantiated using the family selections $\overline{Z}$. The notation $e[X \mapsto Z]$ (read e with X bound to Z) is defined as $e[X \mapsto Z] = e[Z/X, \overline{\gamma}{<}Z{>}/X.\overline{\gamma}', \overline{Z}\ \mathsf{as}\ \overline{\oplus\overline{\gamma}''}/\overline{X}\ \mathsf{as}\ \overline{\oplus\overline{\gamma}''}]$ where: $\mathit{projs}(C, X) = \overline{X}\ \mathsf{as}\ \overline{\oplus\overline{\gamma}''}$ and $(\Phi\ \mathsf{as}\ \oplus\overline{\gamma})\ \mathsf{as}\ \oplus\overline{\gamma}' = \Phi\ \mathsf{as}\ \oplus\overline{\gamma}'$ when $\{\overline{\gamma}'\} \subseteq \{\overline{\gamma}\}$.

Finally, Figure B.16 defines the $\gamma\Phi C_0$ operational semantics. The judgements there are of the scheme $e \to e'$ ("expression e reduces to e' in one step"). There are three

$$\frac{\mathit{fpub}(C, X) = \oplus\overline{\gamma} \qquad \mathit{fpub}(C', X') = \oplus\overline{\gamma}' \qquad \overline{\gamma} \overset{s}{\equiv} \overline{\gamma}'}{C \vdash \mathit{ssfp}(X, X', C')\ \text{ok}}\ (\text{S-WFP})$$

$$\frac{\mathit{valid\text{-}as}(C, X, \oplus\overline{\gamma}) \qquad \mathit{fpub}(C', X') = \oplus\overline{\gamma}' \qquad \overline{\gamma} \overset{s}{\equiv} \overline{\gamma}'}{C \vdash \mathit{ssfp}(X\ \mathsf{as}\ \oplus\overline{\gamma}, X', C')\ \text{ok}}\ (\text{S-PFP})$$

$$\frac{\begin{array}{c}\mathit{fpub}(C', X') = \oplus\overline{\gamma}' \qquad \mathit{cls\text{-}ccomb}(C, S) = \oplus\overline{\gamma} \\ \mathit{fp}(S) = X \qquad \mathit{projs}(C', X') = \overline{X'}\ \mathsf{as}\ \overline{\oplus\overline{\gamma}''}\end{array}}{R[C \xrightarrow{[S/X']} C'] = R[X/X', X.\overline{\gamma}/X'.\overline{\gamma}', \overline{S}\ \mathsf{as}\ \overline{\oplus\overline{\gamma}''}/\overline{X'}\ \mathsf{as}\ \overline{\oplus\overline{\gamma}''}]}\ (\text{FC})$$

Figure A.12: Typing Utility Functions

$$\boxed{\Gamma; G \vdash e : R}$$

$$\Gamma; G \vdash x : \Gamma(x) \qquad (\text{T-Var})$$

$$\frac{\Gamma; \gamma \vdash e_\gamma : R \qquad \mathit{fields}(R) = \overline{R}\ \overline{f}}{\Gamma; \gamma \vdash e_\gamma.f_i : R_i}\ (\text{T-Field})$$

$$\frac{\begin{array}{ll}\Gamma; G \vdash e : X.\gamma & \mathit{mtype}(m_\gamma, \gamma) = \overline{R} \to R \\ \Gamma; G \vdash \overline{e} : \overline{R'} & G \vdash \overline{R'} <: \overline{R}\end{array}}{\Gamma; G \vdash e.m_\gamma(\overline{e}) : R}\ (\text{T-Invk}_1)$$

$$\frac{\begin{array}{ll}G \vdash X.\gamma\ \text{ok} & \mathit{fields}(X.\gamma) = \overline{R} \\ \Gamma; G \vdash \overline{e} : \overline{R'} & G \vdash \overline{R'} <: \overline{R}\end{array}}{\Gamma; G \vdash \mathsf{new}\ X.\gamma(\overline{e}) : X.\gamma}\ (\text{T-New})$$

$$\frac{\mathit{mtype}(m_C, C) = \overline{R} \to R \qquad \Gamma; C \vdash \overline{e} : \overline{R'} \qquad C \vdash \overline{R'} <: \overline{R}}{\Gamma; C \vdash m_C(\overline{e}) : R}\ (\text{T-Invk}_2)$$

$$\frac{\begin{array}{c}\Gamma; C \vdash \overline{e} : \overline{R_e} \qquad \mathit{fps}(C') = \overline{X'} \qquad \#\overline{X'} = \#\overline{S} \\ C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')\ \text{ok} \qquad \mathit{mtype}(m_C, C') = \overline{R'} \to R' \\ \overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C'] \qquad C \vdash \overline{R_e} <: \overline{R} \qquad R = R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']\end{array}}{\Gamma; C \vdash C'\langle\overline{S}\rangle.m_C(\overline{e}) : R}\ (\text{T-Invk}_3)$$

Figure A.13: Expression Typing in $\gamma\Phi C_0$

$$\frac{\begin{array}{c}\overline{D_\Phi}\ \text{ok} \qquad I\ \text{ok} \qquad \mathit{mtype}(m_C, I) = \overline{A} \to A \\ \overline{Z} = \mathit{fsel}(\overline{v}) \qquad I = C\langle\overline{Z'}\rangle \qquad \overline{Z} = \overline{Z'} \\ \mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{v} : \overline{A'} \qquad \mathit{as\text{-}fam}(\overline{Z}) \vdash \overline{A'} <: \overline{A}\end{array}}{\overline{D_\Phi}; I.m_C(\overline{v}) : A}\ (\text{T-Test})$$

Figure A.14: The Test Typing of $\gamma\Phi C_0$

reduction rules in Figure B.16: one for field access, one for component method invocation, and one for client method invocation. Note that the reduction rules may be applied at any point in an expression. Hence, $\gamma\Phi C_0$ operational semantics also offers the obvious congruence rules. The rules in Figure B.16 are standard and we drop further explanation.

$$\frac{G\{\cdots \overline{M} \cdots\} \qquad R\ m(\overline{R}\ \overline{x})\{\mathsf{return}\ e\mathsf{;}\} \in \overline{M}}{\mathit{mbody}(m, G) = (\overline{x}, e)}\ (\text{MB-REL})$$

$$\frac{\mathit{mbody}(m, G) = (\overline{x}, e) \qquad \overline{X} = \mathit{fps}(G) \qquad \mathit{fpub}(G, \overline{X}) = \overline{\oplus\overline{\gamma}'} \qquad \mathit{cmixs}(\overline{Z}) = (\_, \overline{\oplus\overline{\gamma}})}{\mathit{mbody}(m, G\text{<}\overline{Z}\text{>}) = (\overline{x}, e[\overline{X} \mapsto \overline{Z}])}\ (\text{MB-ABS})$$

Figure B.15: Body of Method m of G

Computation

$$\frac{\mathit{fields}(\gamma\text{<}Z\text{>}) = \_\ \overline{f}}{\mathsf{new}\ \gamma\text{<}Z\text{>}(\overline{e}).f_i \to e_i}\ (\text{R-FIELD})$$

$$\frac{\mathit{mbody}(m_\gamma, \gamma\text{<}Z\text{>}) = (\overline{x}, e)}{\mathsf{new}\ \gamma\text{<}Z\text{>}(\overline{e}).m_\gamma(\overline{e}') = e[\overline{e}'/\overline{x}, \mathsf{new}\ \gamma\text{<}Z\text{>}(\overline{e})/\mathsf{this}]}\ (\text{R-INVK}_1)$$

$$\frac{\mathit{mbody}(m_C, C\text{<}\overline{Z}\text{>}) = (\overline{x}, e)}{C\text{<}\overline{Z}\text{>}.m_C(\overline{e}) = e[\overline{e}/\overline{x}]}\ (\text{R-INVK}_3)$$

Congruence

$$\frac{e \to e'}{e.f \to e'.f}\ (\text{RC-FIELD}) \qquad \frac{e \to e'}{e.m(\overline{e}) \to e'.m(\overline{e})}\ (\text{RC-INVK}_1)$$

$$\frac{e_i \to e_i'}{e.m(\ldots, e_i, \ldots) \to e.m(\ldots, e_i', \ldots)}\ (\text{RC-INVK}_2)$$

$$\frac{e_i \to e_i'}{C\text{<}\overline{Z}\text{>}.m(\ldots, e_i, \ldots) \to C\text{<}\overline{Z}\text{>}.m(\ldots, e_i', \ldots)}\ (\text{RC-INVK}_3)$$

$$\frac{e_i \to e_i'}{\mathsf{new}\ \gamma\text{<}Z\text{>}(\ldots, e_i, \ldots) \to \mathsf{new}\ \gamma\text{<}Z\text{>}(\ldots, e_i', \ldots)}\ (\text{RC-NEW})$$

Figure B.16: The $\gamma\Phi C_0$ Operational Semantics